The Dissertation Committee for Navid Yaghmazadeh
certifies that this is the approved version of the following dissertation:

Automated Synthesis of Data Extraction and Transformation Programs

Committee:

Isil Dillig, Supervisor

Keshav Pingali

Raymond Mooney

Armando Solar-Lezama

Automated Synthesis of Data Extraction and Transformation Programs

by

Navid Yaghmazadeh

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2017

Dedicated to the memory of my grandma, *Janu*,

who was the symbol of kindness for me.

Also, to my lovely mom and dad.

Acknowledgments

I wish to thank all the people who helped me throughout my Ph.D. journey.

First, I thank my advisor, Isil Dillig, for all of her guidance and support. She has taught me most of what I know today about program synthesis. Her vast knowledge of the subject and her patience in guiding me truly helped me to overcome challenges in my research. I have always been inspired by her enthusiasm for conducting impactful research on fundamental problems. Her influence in my life has not been limited to my research career. She has been a great friend to me whose endless support helped me pull myself through hard times of my graduate student life. I cherish the valuable real-life lessons I have learnt from her.

I would like to thank Alison Norman for the motivation and support she provided me throughout my Ph.D. study. She helped me make the right decisions during the most difficult days of my Ph.D. tenure. Her encouragement has always been a constant boost of energy to me. I also want to thank Lorenzo Alvisi who was my first mentor in graduate school. What I have learnt from him, both technically and personally, have had a great influence in my life.

I want to thank Kostas Ferles, Yu Feng, Yuepeng Wang, Xinyu Wang, and my other friends and lab-mates at the *UToPiA* research group for their

technical and motivational support. I would like to thank all of my collaborators whose insights highly improved the quality of my research. I would also like to thank all members of my committee for their comments and suggestions.

I would like to thank all of my amazing friends who brought happiness to my life. Among them, I want to express my special gratitude to Pegah Rajaei, Hoda Hashemi, Amin Shams, Niloofar Karimipour and Sepehr Dara for their priceless friendship, encouragement and support.

Finally, a special thanks to my family: my father, Jassem, my mother, Sanambar, and my siblings, Mina, Omid and Shima. Words cannot express my gratitude to them for their unconditional love and all of the sacrifices they have made for me. Without them, none of my success would be possible.

Automated Synthesis of Data Extraction and Transformation Programs

Publication No. _____

Navid Yaghmazadeh, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Isil Dillig

Due to the abundance of data in today's data-rich world, end-users increasingly need to perform various data extraction and transformation tasks. While many of these tedious tasks can be performed in a programmatic way, most end-users lack the required programming expertise to automate them and end up spending their valuable time in manually performing various data-related tasks. The field of program synthesis aims to overcome this problem by automatically generating programs from informal specifications, such as input-output examples or natural language.

This dissertation focuses on the design and implementation of new systems for automating important classes of data transformation and extraction tasks. It introduces solutions for automating data manipulation tasks on fully-structured data formats like relational tables, or on semi-structured formats such as XML and JSON documents.

First, we describe a novel algorithm for synthesizing hierarchical data transformations from input-output examples. A key novelty of our approach is that it reduces the synthesis of tree transformations to the simpler problem of synthesizing transformations over the paths of the tree. We also describe a new and effective algorithm for learning path transformations that combines logical SMT-based reasoning with machine learning techniques based on decision trees.

Next, we present a new methodology for learning programs that migrate tree-structured documents to relational table representations from input-output examples. Our approach achieves its goal by decomposing the synthesis task to two subproblems of (A) learning the column extraction logic, and (B) learning the row extraction logic. We propose a technique for learning column extraction programs using deterministic finite automata, and a new algorithm for predicate learning which combines integer linear programing and logic minimization.

Finally, we address the problem of automating data extraction tasks from natural language. Specifically, we focus on data retrieval from relational databases and describe a novel approach for learning SQL queries from English descriptions. The method we describe is fully automatic and database-agnostic (i.e., does not require customization for each database). Our method combines semantic parsing techniques from the NLP community with novel programming languages ideas involving probabilistic type inhabitation and automated sketch repair.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Much of the data that users deal with today are organized with respect to well-defined data models. These data models include fully-structured data formats like relational databases, and semi-structured hierarchical formats such as XML and JSON documents. These structural formats are used by many applications in various domains -including finance, medicine, retail, and more- to store and exchange data. For instance, XML and JSON documents are popular for exporting data and transferring them between different applications because they incorporate both data and meta-data. On the other hand, since accessing data in relational databases does not require navigating through a hierarchy, they are the best fit for applications that frequently perform data extraction queries on large data sources.

End-users of fully- or semi- structured data types often perform various tasks which involve data extraction and transformation. For example, a user may want to reorganize the structure of elements in an XML document, or converting its data to a relational format. These examples require transformation of the data stored in the XML document. Another example, which involves data extraction, is when a user needs to query data stored in some

relational database by using a declarative query languages such as SQL. Many of these tasks are tedious to be performed manually, especially when they are applied to large data sources. In principle, one may accomplish these tasks by writing a program to automate them. However, most end-users often do not have the required expertise to write such programs and end up spending their time in manually performing them. The field of program synthesis aims to overcome this problem by automatically generating programs from informal specifications, such as input-output examples or natural language.

The idea of program synthesis is to automatically find programs in an underlying domain specific language (DSL) that satisfy a given set of constraints expressed in formal or informal specifications [71]. Since providing formal specification of a problem requires domain-specific knowledge, the program synthesis systems that are designed to interact with non-expert end-users rely on informal problem descriptions such as input-output examples or natural language. These systems can be categorized to two main classes of *programming-by-example (PBE)* [117, 75] and *programming-by-natural-language (PBNL)* [46].

Programming by Example (PBE). There are active researches in the field of program synthesis that focus on generating programs from examples [73, 26, 161, 61, 59, 89, 145]. Given a set of input-output examples, the goal of *PBE* systems is to learn programs such that applying them to each input example return its corresponding output example. Such computer-aided programming

techniques have been successful in a wide range of domains, ranging from spreadsheet programming [73, 27, 108], to string manipulation [161], to table-to-table transformations [59, 176, 89, 145]. PBE has emerged as a favorite program synthesis methodology since reasoning about examples is more systematic than other types of informal specification, and providing examples is easy for end-users in many domains.

Programming by Natural Language (PBNL). Although PBE approach is successful in providing solutions for many synthesis problems, it is not effective in cases where constructing examples is difficult, or a large number of examples is required to describe a task. For instance, providing examples to describe a task in the database domain requires the user to be familiar with the underlying database schema. In such scenarios, it is easier for end-users to specify the problem in natural language. Therefore, a wide range of program synthesizers use the *PBNL* approach to generate programs from natural language specifications [46, 113, 78, 47, 149, 148, 110] in variety of contexts including databases, spreadsheets and smartphone automation scripts.

In this dissertation, we show how program synthesis can help non-expert users to automate their data manipulation tasks. We introduce practical methods based on PBE and PBNL approaches that synthesize data extraction and transformation programs on fully- or semi- structured data formats. We also present systems that are developed using these methodologies and empirically demonstrate their practicality and effectiveness.

First, we present a novel algorithm for synthesizing transformations on tree-structured data -such as Unix directories and XML documents- from input-output examples. Chapter 2 describes this technique in details. Our central insight is to reduce the problem of synthesizing tree transformers to the synthesis of list transformations that are applied to the paths of the tree. We also propose a new and effective algorithm for learning path transformations that combines SMT solving and decision tree learning. We implement our method in a tool called HADES and use it to synthesize bash scripts and XSLT programs for various tasks obtained from on-line forums. Our evaluation shows that HADES can generate the desired program for all of these benchmarks in 0.77 seconds on average.

Chapter 3 presents a novel programming-by-example approach, and its implementation in a tool called MITRA, for automatically migrating tree-structured documents to relational tables. We propose a tree-to-table transformation DSL that facilitates synthesis by allowing us to decompose the synthesis task into two subproblems of learning the column extraction logic, and learning the row extraction logic. Moreover, we describe a synthesis technique for learning column transformation programs using deterministic finite automata, and present a predicate learning algorithm that reduces the problem to a combination of integer linear programming and logic minimization. In order to evaluate our approach, we use MITRA to automate 98 data transformation tasks collected from StackOverflow. Our method can generate the desired program for 94% of these benchmarks with an average synthesis time of 3.8

seconds. We also show that MITRA can convert real-world XML and JSON datasets to full-fledged relational databases.

In Chapter 4 we focus on the synthesis of programs that extract data from relational databases. We present a new technique for automatically synthesizing SQL queries from natural language. At the core of our technique is a new NL-based program synthesis methodology that combines semantic parsing techniques from the NLP community with type-directed program synthesis and automated program repair. Starting with a program sketch obtained using standard parsing techniques, our approach involves an iterative refinement loop that alternates between quantitative type inhabitation and automated sketch repair. We use the proposed idea to build an end-to-end system called SQLIZER that can synthesize SQL queries from natural language. Our method is fully automated, works for any database without requiring additional customization, and does not require users to know the underlying database schema. We evaluate our approach on over 450 natural language queries concerning three different databases, namely MAS, IMDB, and YELP. Our experiments show that the desired query is ranked within the top 5 candidates in close to 90% of the cases.

In summary, we show that program synthesis can automate various difficult tasks that deal with fully- or semi- structured data formats. This is especially helpful for end-users without any programming knowledge who interact with large amount of data on a daily basis. In this dissertation, we identify three important classes of data manipulation tasks and develop new

methods that perform these tasks automatically by synthesizing programs from informal descriptions. We show that all of these methods are practical and can be used to solve real-world problems.

Chapter 2

HADES [1]

In this chapter, we introduce HADES, a new system for synthesizing tree transformations from input-output examples. HADES develops a new method for synthesizing transformations on hierarchical data formats, such as Unix directories and XML documents. We evaluate our approach by collecting a variety of interesting transformations collected from online forums and conducting a user study. We show that HADES can efficiently synthesize tree transformation programs for real world tasks.

We start by introducing our new method and motivations behind it in Section 2.1, and then provide an overview of our approach through an example in Section 2.2. Then, we introduce *hierarchical data trees* (Section 2.3) and describe the detail of our synthesis algorithm in Section 2.4 and Section 2.5. In Section 2.6, we discuss our implementation and present the results of our user study in Section 2.7. We finish this chapter with a brief summary in Section 2.8.

[1] Parts of this chapter have appeared in [184].

2.1 Introduction

Much of the data that users deal with today are inherently hierarchical or tree-shaped. Examples of these hierarchical data formats include:

- *File systems:* A file system is naturally seen as a tree where directories represent internal nodes and files correspond to leaves.
- *XML documents:* In XML documents, data is organized as a tree structure, where each subtree is identified by a pair of start and end tags.
- *HDF files:* Many scientific documents are stored as HDF files that have a tree structure. In this format, *groups* correspond to internal nodes and *datasets* represent leaves.

End-users of such hierarchical data must often perform various kinds of *tree transformations* on their data. For instance, consider the following motivating scenarios:

- Given a directory called `Music` with subfolders for different musical genres, a user wants to re-organize her files so that `Music` has subdirectories for different classes of files (e.g., mp3, wma), and each such subdirectory has further subdirectories for genres (Rock, Jazz etc.).
- Given an XML file, a user wants to convert the *name* attribute of a *person* tag (e.g., `<person name=...> ... </person>`) to a nested element within the person tag (e.g., `<person><name>... </name></person>`).

In principle, one may accomplish these tasks by writing a program, such as a Bash or XSLT script. However, given that end users often do not have the expertise to write programs, an attractive alternative is to automatically synthesize such a program from a high-level specification. In particular, synthesis of programs from examples [74, 162] seems like a natural fit for this setting.

Motivated by this context, we propose a new algorithm, and its implementation in a system called HADES, for automatically synthesizing hierarchical data transformations from input-output examples. Our algorithm operates on a general abstraction of hierarchical data, called *hierarchical data trees (HDTs)*, and does not place any restrictions on the depth or fanout of the hierarchy. Our method is able to synthesize a rich class of tree transformations that commonly arise in real-world data manipulation tasks, such as restructuring of the data hierarchy and modification of metadata.

Synthesizing programs over unbounded trees is a difficult problem. In spite of recent attempts [13, 62, 136], the problem lacks a comprehensive solution. For example, so far as we know, no existing technique can synthesize nontrivial alterations to the structure of an input tree.

Our approach to tree transformation synthesis is based on a simple but novel insight: We note that, under natural assumptions, tree transformations can be written as a *composition of transformations over the paths of the tree*. Our algorithm uses this observation to generate code that behaves as follows: (1) Given an input tree T, generate the set of paths in T; (2) apply a list transformation to each path; and (3) combine the transformed paths into a

transformed tree.

An alternative to the above strategy is to generate a transformation that operates directly on the tree. While this competing approach could conceivably generate programs that are more compact than those we produce, our approach provides a practical way to synthesize complex programs that change the *structure* of the input tree. Put another way, our approach trades off the complexity of the synthesized programs for faster and more comprehensive synthesis.

Another novelty of our approach is a new algorithm for synthesizing list transformations that combines SMT solving and decision tree learning. Given a set of input-output lists, our algorithm partitions the examples into unifiable groups, where a *unifier* is a conditional-free program containing loops. Our algorithm learns unifiers using SMT solving and uses decision tree learning to find predicates that differentiate one unifiable subset of examples from the others.

We have implemented our technique in a system called HADES, which provides a language-agnostic backend for synthesizing HDT transformations. In principle, HADES can be used to generate code in any DSL, provided it has been plugged into our infrastructure. Our current implementation provides two DSL front-ends, one for bash scripts (Unix directories), and another for XSLT (XML transformations).

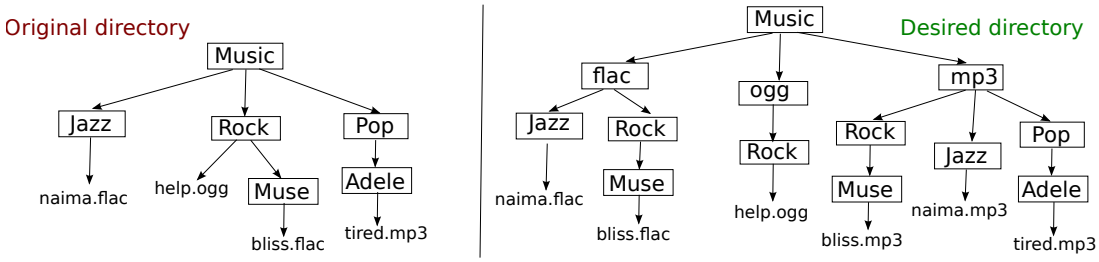


Figure 2.1: HADES' motivating example. Input-output directories for motivating example

2.2 Overview

We illustrate our approach using a motivating example from the file system domain. Consider a user, Bob, who has a large collection of music files organized by genres: The top-level `Music` directory has subfolders for each genre, and each subdirectory contains a collection of music files and subfolders (e.g., one for each band). Furthermore, Bob's music files come in three different formats: mp3, ogg, and flac. However, since not every music player supports all formats, Bob wants to categorize his music based on file type while also maintaining the original organization based on genres. In addition, since few applications can play music files in flac format, Bob wants to convert all his flac files to mp3 and keep both the original as well as the converted files.

Let us consider how Bob can use HADES for synthesizing a bash script that performs his desired transformation. To use HADES, Bob first constructs the input-output example shown in Figure 2.1. Observe that the `Music` directory in the output has three subfolders called `flac`, `ogg`, and `mp3`. Also observe that the `naima.flac` file under the `Jazz` subfolder in the input has

$$
\begin{aligned}
\mathcal{E}_1: \quad & p_1 = [(\text{Music}, d_m), (\text{Jazz}, d_j), (\text{naima}, d_n)] \quad \mapsto \\
& p_1' = [(\text{Music}, d_m), (\text{flac}, d_f), (\text{Jazz}, d_j), (\text{naima}, d_n)] \\
\mathcal{E}_2: \quad & p_1 = [(\text{Music}, d_m), (\text{Jazz}, d_j), (\text{naima}, d_n)] \quad \mapsto \\
& p_1'' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Jazz}, d_j), (\text{naima}, d_n')] \\
\mathcal{E}_3: \quad & p_2 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{help}, d_h)] \quad \mapsto \\
& p_2' = [(\text{Music}, d_m), (\text{ogg}, d_o), (\text{Rock}, d_r), (\text{help}, d_h)] \\
\mathcal{E}_4: \quad & p_3 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_b)] \quad \mapsto \\
& p_3' = [(\text{Music}, d_m), (\text{flac}, d_f), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_b)] \\
\mathcal{E}_5: \quad & p_3 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_b)] \quad \mapsto \\
& p_3'' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_b')] \\
\mathcal{E}_6: \quad & p_4 = [(\text{Music}, d_m), (\text{Pop}, d_p), (\text{Adele}, d_a), (\text{tired}, d_t)] \quad \mapsto \\
& p_4' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Pop}, d_p), (\text{Adele}, d_a), (\text{tired}, d_t)]
\end{aligned}
$$

Figure 2.2: **Path transformation examples constructed by** HADES

been duplicated as `naima.flac` and `naima.mp3` in the output under the `flac/Jazz` and `mp3/Jazz` directories respectively.

Given this input, HADES first converts each of the input and output directories to an intermediate representation called a *hierarchical data tree* (HDT) and then generates a set of list transformation examples $\mathcal{E}$, as shown in Figure 2.2. Each example $e \in \mathcal{E}$ consists of a pair of lists (p_1, p_2) where p_1 is a root-to-leaf path in the input HDT and p_2 is a corresponding path in the output HDT. We represent paths as a list of pairs (l, d) where l is a node label and d is the data stored at that node. In this application domain,

labels correspond to directory/file names, and data includes information about permissions, owner, file type etc.

After constructing the list transformation examples $\mathcal{E}$, HADES synthesizes a *path transformation function* f such that $p' \in f(p)$ for every example $(p, p') \in \mathcal{E}$. Note that we allow path transformers to return a *set* of paths in order to support duplication and deletion.

In the HADES system, the synthesis of path transformers consists of two phases: In the first phase, we *partition* the input examples into sets of *unifiable* groups using SMT solving, and in the second phase, we perform *classification* by using decision tree learning to find a predicate that differentiates one unifiable group from the others. Going back to our example, HADES partitions examples $\mathcal{E}$ into two groups $\mathcal{P}_1 = \{\mathcal{E}_1, \mathcal{E}_3, \mathcal{E}_4, \mathcal{E}_6\}$ and $\mathcal{P}_2 = \{\mathcal{E}_2, \mathcal{E}_5\}$ and infers the following unifier χ_1 for partition $\mathcal{P}_1$:

$$\begin{array}{ll} \text{concat (} & \text{map Id subpath}(x, 1, 1), \\ & \text{map ExtOf subpath}(x, \text{size}(x), \text{size}(x)), \\ & \text{map Id subpath}(x, 2, \text{size}(x)) \quad \text{)} \end{array}$$

Here, $\text{subpath}(x, t, t')$ yields a subpath of x between indices t and t', Id is the identity function, and ExtOf yields the extension (i.e., file type) for a given node. Similarly, for partition $\mathcal{P}_2$, we infer the following unifier χ_2, where FlacToMp3 is a function for converting flac files to mp3:

$$\begin{array}{ll} \text{concat (} & \text{map Id subpath}(x, 1, 1), \text{ “mp3”}, \\ & \text{map Id subpath}(x, 2, \text{size}(x) - 1), \\ & \text{map FlacToMp3 subpath}(x, \text{size}(x), \text{size}(x)) \quad \text{)} \end{array}$$

```
srcDir=$1
for inputFile in \$srcDir/*  do
  elems=$(split $inputFile)
  size=$(SizeOf $elems)
  output=concat($inputElems[0],$(Ext $elems[$size-1]),
                subList(1, $size-1, $elems))
  outputPaths+=\$output
  if {[[ $(Ext $inputFile) == flac]]} then
    output=concat($elems[0], "mp3",
           subList(1, $size-2, $elems),
           $(convertFormat $elems[$size-1]))
    outputPaths+=$output
  fi
done
makeDirectories $outputPaths
```

Figure 2.3: Bash script synthesized by HADES

Next, HADES performs classification to infer a predicate characterizing the input paths in each partition. Since the input paths in partition $\mathcal{P}_1$ include all paths in the input tree, HADES infers the classifier $\phi_1 : \mathit{true}$ for $\mathcal{P}_1$. On the other hand, since partition $\mathcal{P}_2$ only includes p_1, p_3, HADES infers $\phi_2 :$ ext = "flac" as a classifier for $\mathcal{P}_2$. Hence, the overall path transformer inferred by our method is $\pi : \lambda x.\ (\chi_1;\ \text{if(ext = "flac") then } \chi_2)$.

As a final step, HADES uses this list transformer π to synthesize the tree transformation shown as (pseduo-) bash code in Figure 2.3. In essence, the synthesized program constructs the output directory by applying transformation π to every path in the input directory. Going back to our motivating scenario, Bob can now apply this bash script to his very large music collection and obtain the desired transformation.

2.3 Preliminaries

In this section, we define hierarchical data trees and some their properties which we use throughout this chapter.

2.3.1 Hierarchical Data Trees

First, we introduce *hierarchical data trees (HDT)* which our system uses as the canonical representation for various kinds of hierarchical data.

Definition 2.1. Hierarchical data trees (HDT) *Assume a universe* Id *of* labels *for tree nodes and a universe* Dat *of* data. *A* hierarchical data tree T *is a rooted tree represented as a quadruple* (V, E, L, D) *where* V *is a set of nodes, and* E *is a set of directed edges. The* labeling function $L : V \to \text{Id}$ *assigns a label to each node* $v \in V$*, and the* data store $D : V \to \text{Dat}$ *maps each node* $v \in V$ *to the data associated with* v.

We emphasize that the labeling function L does not need to be one-to-one. That is, it is possible that $L(v) = L(v')$ for two distinct nodes v, v'. We write $L(V)$ to indicate the multi-set $\{\ell \mid v \in V \wedge L(v) = \ell\}$ and $L(E)$ to denote the multi-set $\{(\ell, \ell') \mid (v, v') \in E \ \wedge \ L(v) = \ell \wedge L(v') = \ell'\}$.

Example 2.1. *File system directories can be viewed as HDTs where vertices are files and directories, and an edge from* v_1 *to* v_2 *means that* v_1 *is* v_2*'s parent directory. The label for each node* v *is the name of the file or directory associated with* v*. The data store* D *assigns each node to its corresponding meta-data (e.g., permissions, creation date etc.).*

Example 2.2. *We can view XML files as HDTs where nodes correspond to XML elements. An edge from v to v' means that v' is nested directly inside element v. The labeling function L maps each element v to a label (s, i) where s is the name of the tag associated with v and i indicates that v is the i'th element with tag s under v's parent. The data store D maps each element v to its attributes.*

2.3.2 Properties of Hierarchical Data Trees

Next, we define some properties of HDTs.

Definition 2.2. (Well-formedness) *We say that an HDT is* well-formed *iff no two sibling vertices have the same label.*

Throughout this chapter, we assume that HDTs are well-formed and use the term "tree" to mean a well-formed HDT. This well-formedness assumption is a lightweight restriction that applies to many real-world domains. For example, file system directories satisfy the well-formedness assumption because there cannot be two files or directories with the same name under the same directory. XML documents also satisfy this assumption because the order in which tags appear in a document is significant; hence, we can assign two different labels to sibling elements with the same tag name (recall the labeling function from Example 2.2).

Definition 2.3. (Path) *A* path *p in an HDT $T = (V, E, L, D)$ is a list $[(\ell_1, d_1), \ldots, (\ell_k, d_k)]$ such that:*

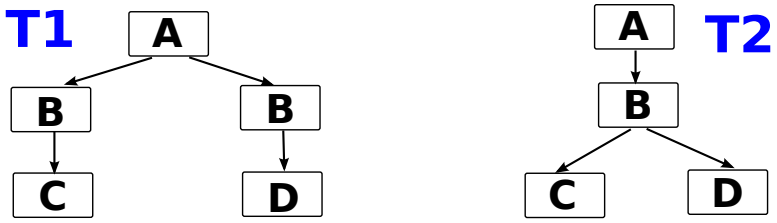


Figure 2.4: Well-formedness in HDTs. An example to motivate well-formedness

- $\ell_1 = L(r)$ *and* $d_1 = D(r)$*, where* r *is the root of* T
- $\ell_k = L(v)$ *and* $d_k = D(v)$*, where* v *is a leaf of* T
- *For each* $i \in [1, k)$*, there is an edge* $(v, v') \in E$ *where* $L(v) = \ell_i$*,* $L(v') = \ell_{i+1}$*,* $D(v) = d_i$*, and* $D(v') = d_{i+1}$.

Given a path $p = [(\ell_1, d_1), \ldots, (\ell_k, d_k)]$, we write $p[i].\ell$ and $p[i].d$ to indicate ℓ_i and d_i respectively. The set of paths in T is denoted by $paths(T)$, and we write $pathTo(T, v)$ to denote a path starting at T's root and ending in v.

Definition 2.4. (Equivalence) *Let* $T_1 = (V_1, E_1, L_1, D_1)$ *with root* v_1 *and* $T_2 = (V_2, E_2, L_2, D_2)$ *with root node* v_2*. We say that* T_1 *is* equivalent to T_2*, written* $T_1 \equiv T_2$*, iff the following conditions hold:*

1. $L_1(v_1) = L_2(v_2)$ *and* $D_1(v_1) = D_2(v_2)$
2. $L_1(\text{children}(v_1)) = L_2(\text{children}(v_2))$
3. *For every* $(v_1', v_2') \in \text{children}(v_1) \times \text{children}(v_2)$ *such that* $L_1(v_1') = L_2(v_2')$*,* $\text{subtree}(T_1, v_1') \equiv \text{subtree}(T_2, v_2')$.

Intuitively, two HDTs T_1 and T_2 are equivalent if they are indistinguishable with respect to the labeling functions (L_1, L_2) and data stores (D_1, D_2). A very important property of *well-formed* trees is that their equivalence can also be stated in terms of paths:

Theorem 2.1.[2] *Let* $T = (V, E, L, D)$ *and* $T' = (V', E', L', D')$ *be two well-formed hierarchical data trees. Then,* $T \equiv T'$ *if and only if* $\text{paths}(T) = \text{paths}(T')$.

This theorem states that a given set of paths uniquely defines a well-formed HDT. This property is very important for our approach since our synthesized programs construct the output tree from a set of paths. However, as illustrated by the following example, this property does not hold if we lift the well-formedness assumption:

Example 2.3. *Consider the HDTs of Figure 2.4, where letters indicate node labels, and assume all nodes store data* d. *In this case, we have* $\text{paths}(T_1) = \text{paths}(T_2)$, *but the left tree* T_1 *is not well-formed, as* A *has two children with label* B.

2.4 Synthesizing Trees from Paths

In this section, we describe our algorithm for synthesizing HDT transformations given an appropriate path transformer.

[2]Proofs of all theorems are given in Appendix A.

Algorithm 2.1 High-level structure of HADES synthesis algorithm

```
1: procedure SYNTHESIZE(set ⟨Tree, Tree⟩ ℰ)
2:     Input: Examples ℰ, consisting of pairs of HDTs
3:     Output: Synthesized program P
4:     if (!CHECKEXAMPLES(ℰ)) then return ⊥;
5:     ℰ′ := FURCATE(ℰ);
6:     f  := INFERPATHTRANS(ℰ′);
7:     P := CODEGEN(f);
8:     return P;
```

2.4.1 Synthesis Algorithm Overview

The high-level structure of our synthesis algorithm consists of four steps and is summarized in Algorithm 2.1. First, given a set of input-output HDTs $\mathcal{E}$, we verify that examples $\mathcal{E}$ obey a certain *unambiguity* restriction required by our algorithm and enforced using the CHECKEXAMPLES function at line 4. Next, we *furcate* the input-output trees $\mathcal{E}$ into a set $\mathcal{E}'$ of *path transformation examples*. Specifically, each example $e \in \mathcal{E}'$ maps a path p in input tree T to a "corresponding" path p' in T' for some $(T, T') \in \mathcal{E}$. Next, we invoke a function called INFERPATHTRANS to learn an appropriate path transformer f such that $p' \in f(p)$ for every $(p, p') \in \mathcal{E}'$. Finally, CODEGEN generates a program that performs the desired tree transformation by applying f to each path in the input tree and then constructing the output tree from the new set of paths. In what follows, we explain these steps in more detail, leaving the INFERPATHTRANS procedure to Section 2.5.

2.4.2 Requirements on Examples

Our approach is parametric on a notion of *correspondence* between paths in the input and output trees. Let Π be the universe of paths in all possible HDTs. A correspondence relation is a binary relation $\sim \subseteq \Pi \times \Pi$. Given a set of input-output examples $\mathcal{E}$, let us define $\mathcal{E}.in, \mathcal{E}.out$ to be the input and output trees in $\mathcal{E}$ respectively. Our synthesis algorithm expects the user-provided examples $\mathcal{E}$ to obey a certain semantic *unambiguity* criterion:

> ***Unambiguity:*** For every $p' \in paths(\mathcal{E}.out)$, there exists a *unique* p such that $p \sim p'$ where $p \in paths(\mathcal{E}.in)$ and $(p, p') \in paths(T) \times paths(T')$ for some $(T, T') \in \mathcal{E}$.

In other words, unambiguity requires that, for every output path p', we can find exactly one input path p such that $p \sim p'$ and p, p' belong to the same input-output example. Unambiguity is enforced by the CheckExamples function used at line 4 of the Synthesize algorithm. [3]

The correspondence relation $\sim$ can be defined in many natural ways. Specifically, the Hades system allows the user to mark paths in the input-output examples as corresponding. However, Hades also comes with a default definition of $\sim$ that is adequate in many practical settings.

[3] We can actually drop the unambiguity requirement by adding another layer of search to the synthesis algorithm. However, we have not encountered any examples that violate this restriction in practice.

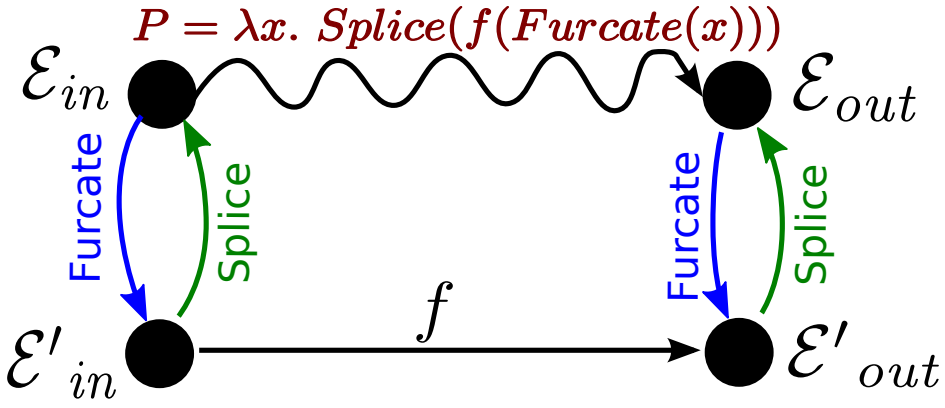


Figure 2.5: Schematic illustration of HADES synthesis approach

2.4.3 Furcation

Given an unambiguous set of examples $\mathcal{E}$, our algorithm *furcates* them into a set of *path transformation examples* $\mathcal{E}'$. Specifically, a pair of paths $(p, p') \in \mathcal{E}'$ iff $p \in \mathit{paths}(T), p' \in \mathit{paths}(T')$, and $p \sim p'$ for some $(T, T') \in \mathcal{E}$. If some input path $p \in \mathit{paths}(T)$ does not have a corresponding output path, then $\mathcal{E}'$ also contains $(p, \perp)$.

Note that $\mathcal{E}'$ may contain multiple examples that have the same path p as an input. For instance, when some leaf in the input tree has been duplicated in the output tree, then there will be at least two examples (p, p') and (p, p'') in $\mathcal{E}'$. However, due to the unambiguity requirement, it is not possible that there are multiple examples in $\mathcal{E}'$ that have the same output path. That is, if $(p, p') \in \mathcal{E}'$, then there does not exist another example $(p'', p') \in \mathcal{E}'$ where $p \neq p''$.

Given path transformation examples $\mathcal{E}'$, we write $\mathit{inputs}(\mathcal{E}')$ and $\mathit{outputs}(\mathcal{E}')$

to denote the input and output paths in $\mathcal{E}'$ respectively. That is, $p \in \mathit{inputs}(\mathcal{E}')$ iff $(p, \_) \in \mathcal{E}'$.

Example 2.4. *Figure 2.2 shows the result of furcating the input-output example from Figure 2.1.*

2.4.4 Path Transformer

The next step in our synthesis algorithm is to learn a *path transformer* that takes an input path and returns a *set* of output paths. Any path transformer f returned by InferPathTrans at line 6 of the Synthesize procedure must satisfy the following requirement: [4]

$$\forall p \in \mathit{inputs}(\mathcal{E}').\ (p' \in f(p) \Leftrightarrow (p, p') \in \mathcal{E}')$$

When an input path p does not have a corresponding output path p', we require that $f(p) = \{\bot\}$. Since InferPathTrans is the most involved aspect of the synthesis algorithm, we discuss it in detail in Section 2.5.

2.4.5 Code Generation

Once we learn a path transformer f, the last step of our algorithm is to generate code for the synthesized program. For this purpose, we first define a *splicing operation*: Given a set S of paths, Splice(S) yields a well-formed tree T such that $\mathit{paths}(T) = S$. Recall from Theorem 2.1 that the result of splicing is unique. Using this splicing operation, the CodeGen procedure used at line

[4] Proof of this property is given in Appendix A.

7 of Figure 2.1 yields the following function P:

$$P = \lambda T.\ \textsc{Splice}(\{p' \mid p' \in f(p) \wedge p \in \mathit{paths}(T) \wedge p' \neq \bot\})$$

In other words, synthesized program P constructs the output tree by applying function f to each path in the input tree.

Summary. Figure 2.5 gives a schematic summary our approach: Our *synthesis algorithm* furcates the input-output examples and learns a path transformer f. On the other hand, the *synthesized algorithm* furcates the input tree, applies path transformer f, and splices it back to obtain the output tree.

Theorem 2.2. (Soundness) *Let $\mathcal{E}$ be a set of examples satisfying the unambiguity requirement, and let $\mathcal{E}'$ be the output of* $\textsc{Furcate}(\mathcal{E})$*. Then, $\forall (T, T') \in \mathcal{E}$. $P(T) \equiv T'$ where P is the output of procedure* Synthesize *from Figure 2.1.*

2.5 Synthesizing Path Transformations

We now describe the InferPathTrans algorithm for learning path transformers from path examples $\mathcal{E}$. Each example $(p, p') \in \mathcal{E}$ consists of an input path p and an output path p', and our goal is to learn a path transformer f satisfying the property $\forall p \in \mathit{inputs}(\mathcal{E}).(p' \in f(p) \Leftrightarrow (p, p') \in \mathcal{E})$.

2.5.1 Domain Specific Language (DSL)

We first introduce a small language over which we describe path transformers. As shown in Figure 2.6, a path transformer π takes as input a path x

$$
\begin{array}{lcl}
\text{Path transformer } \pi & := & \lambda x.\ \{\phi_1 \to \chi_1 \oplus \ldots \oplus \phi_n \to \chi_n\} \\
\text{Path term } \chi & := & \text{concat}(\tau_1(x), \ldots, \tau_n(x)) \\
\text{Segment trans. } \tau & := & \lambda x.\ \text{map } F\ \text{subpath}(x, t_1, t_2) \\
\text{Index term } t & := & b \cdot \text{size}(x) + c \\
\text{Path cond } \phi & := & P_i(x) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \\
\text{Mapper } F & := & \lambda x.\ \text{int} \mid \lambda x.\ f_i(x) \\
 & & \mid \lambda x.\ \text{if}(\varphi_i(x))\ \text{then } f_i(x)\ \text{else } f_j(x)
\end{array}
$$

Figure 2.6: Language for expressing path transformers

and returns a set of paths $\Pi = \{p_1, \ldots, p_k\}$. Specifically, a path transformer π has the syntax $\lambda x.\ \{\phi_1 \to \chi_1 \oplus \ldots \oplus \phi_n \to \chi_n\}$ where each ϕ_i is a *path condition* that evaluates to true or false, and each χ_i is a *path term* describing an output path. The semantics of this construct is that $\chi_i \in \Pi$ iff ϕ_i evaluates to true. We refer to the number of (ϕ_i, χ_i) pairs in π as the *arity* of π.

Path terms χ used in the path transformer are formed by concatanating different subpaths $\tau_i(x)$ where each τ_i is a so-called *segment transformer*. A segment transformer τ is of the form $\lambda x.\text{map } F\ \text{subpath}(x, t_1, t_2)$ and applies function F to a subpath of x starting at index t_1 and ending at index t_2 (inclusive). For brevity, we often abbreviate $\lambda x.\text{map } F\ \text{subpath}(x, t_1, t_2)$ using the notation $\langle t_1, t_2, F\rangle$. Note that a segment transformer is a kind of looping construct that iterates over a consecutive range of elements in x. Indices in segment transformers are specified using *index terms* t of the form $b \cdot \text{size}(x) + c$ where b is either 0 or 1, c is an integer, and $\text{size}(x)$ denotes the number of

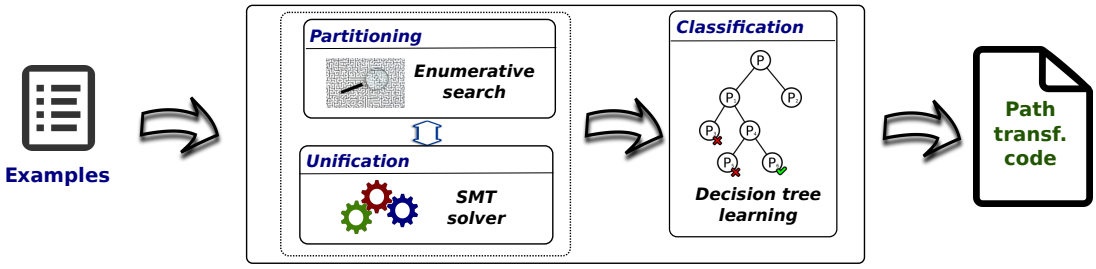


Figure 2.7: Schematic overview of algorithm for learning path transformers

elements in path x. [5] Mapper functions F either return constants or apply pre-defined functions f_i to their input. For instance, in the file system domain, such predefined functions include procedures for changing file permission or converting one file type to another (e.g., jpg to png). Mapper functions F can also contain if statements if$(\varphi_i(x))$ then $f_i(x)$ else $f_j(x)$ where each φ_i is drawn from a family of pre-defined predicate templates (e.g., for checking file type).

2.5.2 Learning Path Transformers

We now give an overview of our algorithm for learning path transformers. As illustrated in Figure 2.7, our algorithm consists of three key components, namely *partitioning*, *unification*, and *classification*. The goal of partitioning is to divide examples $\mathcal{E}$ into groups of *unifiable subsets*. We say that a set of examples $\mathcal{E}^*$ is *unifiable* if *outputs*$(\mathcal{E}^*)$ can be represented using the same path term χ^*, and we refer to χ^* as the *unifier* for $\mathcal{E}^*$. Our algorithm represents each partition $\mathcal{P}_i$ as a triple $\langle \mathcal{E}_i, \chi_i, \phi_i \rangle$ where $\mathcal{E}_i$ is a unifiable set of examples, χ_i is

[5] Our implementation also allows terms containing indexOf(x, e) expressions; however, we ignore them here to simplify the presentation.

their unifier, and ϕ_i is a predicate distinguishing $\mathcal{E}_i$ from the other examples.

The partitioning component of our algorithm is based on enumerative search that tries different hypotheses in increasing order of complexity. Here, a *hypothesis* corresponds to a partitioning of examples $\mathcal{E}$ into k disjoint groups $\mathcal{E}_1, \ldots, \mathcal{E}_k$. Given a hypothesis, we query whether each $\mathcal{E}_i$ is unifiable. If unification fails, we backtrack and try a different hypothesis.

Since our method repeatedly invokes the unification algorithm to confirm or refute a hypothesis, we need an efficient mechanism for finding unifiers. Towards this goal, our algorithm represents each input-output example using a compact numeric representation and invokes an SMT solver to determine the existence of a unifier. Furthermore, we can obtain the unifier χ_i associated with examples $\mathcal{E}_i$ by getting a satisfying assignment to an SMT formula. This approach allows our algorithm to find unifiers with a single SMT query rather than explicitly exploring search spaces of exponential size.

The last key ingredient of our algorithm for synthesizing path transformers is *classification*. Given a set of examples $\mathcal{E}_1, \ldots, \mathcal{E}_k$, the goal of classification is to infer a predicate ϕ_i for each $\mathcal{E}_i$ such that ϕ_i evaluates to *true* for each $p \in \mathit{inputs}(\mathcal{E}_i)$ and evaluates to *false* for each $p' \in \mathit{inputs}(\mathcal{E}) - \mathit{inputs}(\mathcal{E}_i)$. For this purpose, we use the ID3 algorithm for learning a small decision tree and then extract a formula describing all positive examples in this tree.

InferPathTrans algorithm. Algorithm 2.2 presents the InferPathTrans procedure based on this discussion. The algorithm consists of two phases: In

Algorithm 2.2 Algorithm for learning path transformations

```
1: procedure InferPathTrans(set 𝓔)
2:     Input: A set of path transformation examples 𝓔
3:     Output: Synthesized path transformer
4:                                          ▷ Phase I: Partition into unifiable subsets
5:     for i=1; i≤ |𝓔|; i++   do
6:         Φ := Partition(∅, 𝓔, i);
7:         if Φ ≠ ∅ then break;
8:                                          ▷ Phase II: Learn classifiers
9:     for all 𝒫_i in Φ do
10:        𝒫_i.φ := Classify(𝒫_i.𝓔, 𝓔);
11:    π := PtCodeGen(Φ);
12:    return π;
```

the first phase, we partition examples $\mathcal{E}$ into a *smallest* set Φ of unifiable groups, and, in the second phase, we infer classifiers for each partition. Specifically, lines 5-7 try to partition $\mathcal{E}$ into i disjoint groups by invoking the Partition procedure, and lines 9-10 infer classifiers. Finally, we use a procedure called PtCodeGen to generate a path transformer π from the partitions in the expected way. In what follows, we describe partitioning, unification, and classification in more detail.

2.5.3 Partitioning

Algorithm 2.3 shows the partitioning algorithm used in InferPathTrans. The recursive Partition procedure takes as input a set of examples $\mathcal{E}_1$ that are part of the same partition, the remaining examples $\mathcal{E}_2$, and number

Algorithm 2.3 Partitioning Algorithm. The notation $\mathcal{P}(\mathcal{E}, \chi)$ indicates a partition with examples $\mathcal{E}$ and their unifier χ.

```
1:  procedure PARTITION(set E1, set E2, int k)
2:      Input: Current partition E1, remaining examples E2,
3:             and number of partitions k
4:      Output: Set of partitions Φ
5:                                                        ▷ Base case
6:      if k = 1 then
7:          χ := UNIFY(E1 ∪ E2);
8:          if χ = null then return ∅;
9:          return {P(E1 ∪ E2, χ)};
10:                                                       ▷ Recursive case
11:     for all e ∈ E2 do
12:         χ := UNIFY(E1 ∪ {e});
13:         if χ = null then continue;
14:         Φ := PARTITION(∅, E2 − {e}, k − 1);
15:         if Φ ≠ ∅ then
16:             return Φ ∪ {P(E1 ∪ {e}, χ)};
17:         Φ := PARTITION(E1 ∪ {e}, E2 − {e}, k);
18:         if Φ ≠ ∅ then return Φ;
19:     return ∅;
```

of partitions k. The base case of the algorithm is when $k = 1$: In this case, we try to unify all examples in $\mathcal{E}_1 \cup \mathcal{E}_2$, and, if this is not possible, we return failure (i.e., $\emptyset$).

In the recursive case (lines 11–18), we try to grow the current partition $\mathcal{E}_1$ by adding one or more of the remaining examples from $\mathcal{E}_2$. The algorithm always maintains the invariant that elements in $\mathcal{E}_1$ are unifiable. Hence, we try to add an element $e \in \mathcal{E}_2$ to $\mathcal{E}_1$ (line 12), and if the resulting set is not

unifiable, we give up and try a different element (line 13). Since $\mathcal{E}_1 \cup \{e\}$ is unifiable, we now check if it is possible to partition the remaining examples $\mathcal{E}_2 - \{e\}$ into $k-1$ unifiable sets (recursive call at line 14). If this is indeed possible, we have found a way to partition $\mathcal{E}_1 \cup \mathcal{E}_2$ into k different partitions and return success (line 16).

Now, if the remaining examples $\mathcal{E}_2$ cannot be partitioned into $k-1$ unifiable sets, we try to shrink $\mathcal{E}_2$ by growing $\mathcal{E}_1$. Hence, the recursive call at line 17 looks for a partitioning of examples where one of the partitions contains *at least* $\mathcal{E}_1 \cup \{e\}$. If this recursive call also does not succeed, then we move on and consider the scenario where partition $\mathcal{E}_1$ does not contain the current element e.

Observe that PARTITION$(\emptyset, \mathcal{E}, k)$ effectively explores all possible ways to partition examples $\mathcal{E}$ into k unifiable subsets. However, since most subsets of $\mathcal{E}$ are typically not unifiable, the algorithm does not come anywhere near its worst-case $O(k^n)$ behavior in practice.

2.5.4 Unification

We now describe the UNIFY procedure for determining if examples $\mathcal{E}$ have a unifier. Since the unification algorithm is invoked many times during partitioning, we need to ensure that UNIFY is efficient in practice. Hence, we formulate it as a symbolic constraint solving problem rather than performing explicit search. However, in order to reduce unification to SMT solving, we first need to represent each input-output example in a so-called *summarized form*

that uses a numerical representation to describe each path transformation.

Intuitively, a *summarized example* represents a path transformation as a *permutation* of the elements in the input path. For example, if some element e in the output path has the same label as the k'th element in the input path, then we represent e using numerical value k. On the other hand, if element e does not have a corresponding element with the same label in the input path, summarization uses a so-called *"dictionary"* $\mathcal{D}$ to map e to a numerical value. More formally, we define example summarization as follows:

Definition 2.5. (Example summarization) *Let $\mathcal{E}$ be a set of examples where $\mathcal{L}$ denotes the labels used in $\mathcal{E}$, and let $\mathcal{F}$ be the set of pre-defined functions allowed in the path transformer. Let $\mathcal{D} : (\mathcal{L} \cup \mathcal{F}) \to \{i \mid i \in \mathbb{Z} \wedge i > m\}$ be an injective function where m is the maximum path length in* inputs($\mathcal{E}$)*. Given an example $(p_1, p_2) \in \mathcal{E}$, the summarized form of (p_1, p_2) is a pair (n, σ) where n is the length of path p_1 and σ is a sequence such that:*

$$\sigma_i : \begin{cases} (j, \quad p_1[j].\mathrm{d} \to p_2[i].\mathrm{d}) & \text{if } \exists j.p_2[i].\ell = p_1[j].\ell \\ (\mathcal{D}(f^*) + j, \quad \bot \to p_2[i].\mathrm{d}) & \text{else if } \exists j.p_2[i].\ell = f^*(p_1[j]) \\ (\mathcal{D}(p_2[i].\ell), \quad \bot \to p_2[i].\mathrm{d}) & \text{otherwise} \end{cases}$$

We illustrate summarization using a few examples:

Example 2.5. *Consider input path $p1 = [(A, r), (B, r), (C, r)]$, and output path $p_2 = [(C, w), (A, r), (B, r)]$, where r, w indicate permissions. The summarized example is $(3, \sigma)$ where $\sigma = [(3, r \mapsto w), (1, r \mapsto r), (2, r \mapsto r)]$. The first element in σ is $(3, r \mapsto w)$ because the first element C of the output path is at index 3 in the input path, and its corresponding data is mapped from r to w.*

Example 2.6. *Consider the same p_1 from Example 2.5 and the output path $p'_2 = [(A, r), (B, r), (\text{New}, r)]$. Suppose that $\mathcal{D}(\text{New}) = 1000$ (i.e., "dictionary" assigns 1000 to foreign element* New*). The summarized example is $(3, \sigma')$ where $\sigma' = [(1, r \mapsto r), (2, r \mapsto r), (1000, \bot \mapsto r)]$.*

Example 2.7. *Consider the input path $[(A, \bot), (B, pdf)]$ and output $[(A, \bot), (\text{pdf}, \bot), (B, pdf)]$. In this case, the summarized example is $(2, \sigma)$ where $\sigma = [(1, \bot \mapsto \bot), (\mathcal{D}(\text{ExtOf}) + 2, \bot \mapsto \bot), (2, pdf \mapsto pdf)]$. Note that label pdf in the output list is mapped to $\mathcal{D}(\text{ExtOf}) + 2$ because it corresponds to the extension for element at index 2 in the input list (case 2 of Definition 2.5).*

Given a summarized example $e = (n, [(i_1, \_), \ldots, (i_n, \_)])$, we write *indices*$(e)$ to denote $[i_1, \ldots, i_n]$. For instance, in Example 2.5, we have *indices*$(e) = [3, 1, 2]$.

The next step in our unification algorithm is to *coalesce* consecutive indices in the summarized example. Hence, we define the *coalesced form* of an example as follows:

Definition 2.6. (Coalesced form) *Given a summarized example $e = (n, \sigma)$, we say that e^* is a* coalesced form *of e iff it is of the form $(n, [\langle b_1, e_1, M_1\rangle, \ldots, \langle b_k, e_k, M_k\rangle])$ where*

- $\text{indices}(e) = [b_1, \ldots, e_1, \ldots, b_k, \ldots, e_k]$
- $\forall i$, $[b_i, b_{i+1}, \ldots, e_i]$ *is a contiguous sublist of* $\text{indices}(e)$

- $M_k = \bigcup_j \{m_j \mid b_k \leq i_j \leq e_k \wedge \sigma_j = (i_j, m_j)\}$

Intuitively, this definition "coalesces" consecutive indices in a summarized example. Note that the coalesced form of an example is *not* unique because we are *allowed* but *not required* to coalesce consecutive indices.

Example 2.8. *Consider the summarized example from Example 2.5, which has the following two coalesced forms:*

$$e_1^* = (3, [\langle 3, 3, \{r \mapsto w\}\rangle, \langle 1, 1, \{r \mapsto r\}\rangle, \langle 2, 2, \{r \mapsto r\}\rangle])$$
$$e_2^* = (3, [\langle 3, 3, \{r \mapsto w\}\rangle, \langle 1, 2, \{r \mapsto r\}\rangle])$$

Given a coalesced example $e^* = (n, \sigma^*)$ where σ^* is $[\langle b_1, e_1, M_1\rangle, \ldots, \langle b_k, e_k, M_k\rangle]$, we define $\mathit{len}(e^*)$ to be n and $\mathit{segments}(e^*)$ to be k. We also write $\mathit{begin}(e^*, j)$ to indicate b_j, $\mathit{end}(e^*, j)$ for e_j, and $\mathit{data}(e^*, j)$ for M_j. Note that σ^* can be viewed as a concatanation of *concrete path segments* of the form $\langle c, c', M\rangle$ where c and c' are the start and end indices for the corresponding path segment respectively.

Before we continue, let us notice the similarity between segment transformers $\langle t, t', F\rangle$ [6] in the language from Figure 2.6, and each concrete path segment $\langle c, c', M\rangle$ in a coalesced example. Specifically, observe that a concrete path segment can be viewed as a *concrete instantiation* of a segment transformer $\langle b * \mathit{size}(x) + c, b' * \mathit{size}(x) + c', F\rangle$ where each of the terms b, c, b', c', and F are substituted by concrete values. In fact, this is no coincidence: *The key insight underlying our unification algorithm is to use the concrete path*

[6] Recall that $\langle t, t', F\rangle$ is an abbreviation for $\lambda x.$ map F subpath(x, t, t').

segments in the coalesced examples to solve for the unknown terms in segment transformers using an SMT solver.

Algorithm 2.4 Unification algorithm

1: **procedure** UNIFY(set $\mathcal{E}$)
2: **Input:** A set of path transformation examples $\mathcal{E}$
3: **Output:** A unifier χ if it exists, null otherwise
4: ▷ Convert examples to coalesced form
5: $\mathcal{E}' := \{e' \mid e' = \textsc{Summarize}(e) \ \wedge \ e \in \mathcal{E}\}$;
6: $\Lambda := \{(e_1^*, \ldots, e_n^*) \mid e_i^* \in \textsc{Coalesce}(e_i') \wedge e_i' \in \mathcal{E}'\}$;
7: ▷ Generate candidate unifiers χ of increasing size
8: **for** k=1 **to** $\mathit{maxSize}(\mathit{outputs}(\mathcal{E}))$ **do**
9: $\tau_i := \langle b_i \cdot \mathit{size}(x) + c_i \ , b_i' \cdot \mathit{size}(x) + c_i' \ , F_i \rangle$;
10: $\chi := \text{concat}(\tau_1(x), \ldots, \tau_k(x))$;
11: ▷ Check if χ unifies some $\mathcal{E}^* \in \Lambda$
12: **for all** $\mathcal{E}^* \in \Lambda$ **do**
13: **if** $(\exists e_i^* \in \mathcal{E}^*.\ \mathit{segments}(e_i^*) \neq k)$ **then**
14: **continue**;
15: ▷ Use SMT solver to check if χ unifies $\mathcal{E}^*$
16: $\varphi_{e^*}^i := \ (b_i \cdot \mathit{len}(e^*) + c_i = \mathit{begin}(e^*, i))$;
17: $\psi_{e^*}^i := \ (b_i' \cdot \mathit{len}(e^*) + c_i' = \mathit{end}(e^*, i))$;
18: $\phi \ := \bigwedge_{1 \leq i \leq k} \bigwedge_{e^* \in \mathcal{E}^*} (\varphi_{e^*}^i \wedge \psi_{e^*}^i)$;
19: **if** UNSAT(ϕ) **then continue;**
20: $\sigma := \textsc{SatAssign}(\phi)$;
21: $\sigma' := \textsc{UnifyMappers}(\mathcal{E}^*)$;
22: **if** σ' = null **then continue**;
23: **return** $\textsc{Substitute}(\chi, \sigma \cup \sigma')$;
24: **return** null;

Let us now consider the unification algorithm presented in Algorithm 2.4. Given examples $\mathcal{E}$, the UNIFY algorithm first computes the summarized examples $\mathcal{E}'$ and then generates all possible coalesced forms (lines 5-6). Since we do not know which coalesced form is the "right" one, we need to consider all possible combinations of coalesced forms of the examples. Hence, set Λ from line 6 corresponds to the Cartesian product of the coalesced form of examples $\mathcal{E}'$.

Next, the algorithm enumerates all possible candidate unifiers χ of increasing arity. Based on the grammar of our language (recall Figure 2.6), a path term χ of arity k has the shape $\text{concat}(\tau_1(x), \ldots, \tau_k(x))$ where each τ_i is a segment transformer of the form $\langle b_i \cdot \mathit{size}(x) + c_i \ , b'_i \cdot \mathit{size}(x) + c'_i \ , F_i \rangle$. Hence, the hypothesis χ at line 10 is a templatized unifier whose unknown coefficients will be inferred later.

Given a hypothesis χ, we next try to confirm or refute this hypothesis by checking if there exists some $\mathcal{E}^* \in \Lambda$ for which χ is a unifier. For χ to be a unifier for $\mathcal{E}^*$, every example $e_i^* \in \mathcal{E}^*$ must contain exactly k segments because χ has arity k. If this condition is not met (line 13), χ cannot be a unifier for $\mathcal{E}^*$, so we reject it.

If all examples in $\mathcal{E}^*$ contain k segments, we try to instantiate the unknown coefficients $b_1, b'_1, c_1, c'_1, \ldots, b_k, b'_k, c_k, c'_k$ in χ in a way that is consistent with the concrete path segments in all examples in $\mathcal{E}^*$. Now, consider the i'th concrete path segment in coalesced example e^* and the i'th abstract path segment $\langle b_i \cdot \mathit{size}(x) + c_i, b'_i \cdot \mathit{size}(x) + c'_i, F_i \rangle$ in hypothesis χ. Clearly, if

$$\mathcal{E}'_1: \quad (3, [(1, d_m \mapsto d_m), (\mathcal{D}(\mathit{ExtOf}) + 3, \bot \mapsto d_f), (2, d_j \mapsto d_j), (3, d_n \mapsto d_n)])$$

$$\mathcal{E}'_3: \quad (3, [(1, d_m \mapsto d_m), (\mathcal{D}(\mathit{ExtOf}) + 3, \bot \mapsto d_o), (2, d_r \mapsto d_r), (3, d_h \mapsto d_h)])$$

$$\mathcal{E}'_4: \quad (4, [(1, d_m \mapsto d_m), (\mathcal{D}(\mathit{ExtOf}) + 4, \bot \mapsto d_f), (2, d_r \mapsto d_r), (3, d_{ms} \mapsto d_{ms}), (4, d_b \mapsto d_b)])$$

$$\mathcal{E}'_6: \quad (4, [(1, d_m \mapsto d_m), (\mathcal{D}(\mathit{ExtOf}) + 4, \bot \mapsto d_{mp}), (2, d_p \mapsto d_p), (3, d_a \mapsto d_a), (4, d_t \mapsto d_t)])$$

Figure 2.8: Summarization of partition $\mathcal{P}_1$ of the motivating example.

our hypothesis is correct, it should be possible to instantiate the unknown coefficients in a way that satisfies:

$$b_i \cdot \mathit{len}(e^*) + c_i = \mathit{begin}(e^*, i) \ \wedge \ b'_i \cdot \mathit{len}(e^*) + c'_i = \mathit{end}(e^*, i)$$

since the size of this example is $\mathit{len}(e^*)$ and begin and end indices for the path segment are $\mathit{begin}(e^*, i)$ and $\mathit{end}(e^*, i)$. Hence, we test the correctness of hypothesis χ for $\mathcal{E}^*$ by querying the satisfiability of formula ϕ from line 18. If ϕ is unsatisfiable (line 19), we reject the hypothesis for $\mathcal{E}^*$.

If, however, ϕ is satisfiable, we have found an instantiation of the unknown coefficients, which is given by the satisfying assignment σ at line 20. Now, the only remaining question is whether we can also find an instantiation of the unknown functions $F_1, \ldots, F_k$ used in χ. For this purpose, we use a function called UnifyMappers which tries to find mapper functions F_i unifying all the different M_i's from the examples. Since the UnifyMappers procedure is based on straightforward enumerative search, we do not describe

$$\mathcal{E}_1^*: \quad (3, [\langle 1, 1, \{d_m \mapsto d_m\}\rangle, \langle \mathcal{D}(\mathit{ExtOf}) + 3, \mathcal{D}(\mathit{ExtOf}) + 3, \{\bot \mapsto d_f\}\rangle, \langle 2, 3, \{d_j \mapsto d_j, d_n \mapsto d_n\}\rangle])$$

$$\mathcal{E}_3^*: \quad (3, [\langle 1, 1, \{d_m \mapsto d_m\}\rangle, \langle \mathcal{D}(\mathit{ExtOf}) + 3, \mathcal{D}(\mathit{ExtOf}) + 3, \{\bot \mapsto d_o\}\rangle, \langle 2, 3, \{d_r \mapsto d_r, d_h \mapsto d_h\}\rangle])$$

$$\mathcal{E}_4^*: \quad (4, [\langle 1, 1, \{d_m \mapsto d_m\}\rangle, \langle \mathcal{D}(\mathit{ExtOf}) + 4, \mathcal{D}(\mathit{ExtOf}) + 4, \{\bot \mapsto d_f\}\rangle, \langle 2, 4, \{d_r \mapsto d_r, d_{ms} \mapsto d_{ms}, d_b \mapsto d_b\}\rangle])$$

$$\mathcal{E}_6^*: \quad (4, [\langle 1, 1, \{d_m \mapsto d_m\}\rangle, \langle \mathcal{D}(\mathit{ExtOf}) + 4, \mathcal{D}(\mathit{ExtOf}) + 4, \{\bot \mapsto d_{mp}\}\rangle, \langle 2, 4, \{d_p \mapsto d_p, d_a \mapsto d_a, d_t \mapsto d_t\}\rangle])$$

Figure 2.9: A set of coalesced examples $\mathcal{E}^*$ for partition $\mathcal{P}_1$

it in detail. In particular, since the language of Figure 2.6 only allows a finite set of pre-defined data transformers f_i and predicates ϕ_i, UnifyMappers enumerates –in increasing order of complexity– all possible functions belonging to the grammar of mapper functions in Figure 2.6.

Example 2.9. *For the motivating example from Section 2.2, our unification algorithm takes the following steps to determine unifier χ_1 for partition $\mathcal{P}_1$: First, we generate the summarized examples shown in Figure 2.8 and construct set Λ. We then consider hypotheses of increasing size and reject those with arity 1 and 2 since all examples contain at least 3 segments. Now, let's consider hypothesis χ of arity 3 and the set of coalesced examples $\mathcal{E}^*$ shown in Figure 2.9. We generate the formula ϕ shown in Figure 2.10 and get a satisfying assignment, which results in the following instantiation of χ:*

$$[\langle 1, 1, F_1\rangle, \langle v, v, F_2\rangle, \langle 2, \text{size}(x), F_3\rangle]$$

$$
\begin{aligned}
\phi : \quad & b_1 \cdot 3 + c_1 = 1 \ \wedge \ b'_1 \cdot 3 + c'_1 = 1 \ \wedge \ b_1 \cdot 4 + c_1 = 1 \ \wedge \ b'_1 \cdot 4 + c'_1 = 1 \ \wedge \\
& b_2 \cdot 3 + c_2 = \mathcal{D}(\mathit{ExtOf}) + 3 \ \wedge \ b'_2 \cdot 3 + c'_2 = \mathcal{D}(\mathit{ExtOf}) + 3 \ \wedge \\
& b_2 \cdot 4 + c_2 = \mathcal{D}(\mathit{ExtOf}) + 4 \ \wedge \ b'_2 \cdot 4 + c'_2 = \mathcal{D}(\mathit{ExtOf}) + 4 \ \wedge \\
& b_3 \cdot 3 + c_3 = 2 \ \wedge \ b'_3 \cdot 3 + c'_3 = 3 \ \wedge \ b_3 \cdot 4 + c_3 = 2 \ \wedge \ b'_3 \cdot 4 + c'_3 = 4
\end{aligned}
$$

Figure 2.10: (Simplified) formula ϕ. To check the satisfiability of hypothesis χ_1 on set $\mathcal{E}^*$.

where $v = \mathcal{D}(\text{ExtOf}) + \text{size}(x)$. *Finally,* UnifyMappers *searches for instantiations of* F_1, F_2, *and* F_3 *satisfying all data mappers in the examples. For* F_1 *and* F_3, *it returns the Identity function, and for,* F_2, *it extracts the function extOf from the segment transformer coefficients. As a result, we obtain the unifier* χ_1 *from Section 2.2.*

2.5.5 Classification

We now consider the last missing piece of our algorithm, namely classification. Given examples $\mathcal{E}$ and partition $\mathcal{P}_i$ with examples $\mathcal{E}_i \subseteq \mathcal{E}$ and unifier χ_i, the goal of classification is to find a predicate ϕ_i such that:

(1) $\forall p \in \mathit{inputs}(\mathcal{E}_i).\ (\phi_i[p/x] \equiv \mathit{true})$
(2) $\forall p \in (\mathit{inputs}(\mathcal{E}) - \mathit{inputs}(\mathcal{E}_i)).\ (\phi_i[p/x] \equiv \mathit{false})$

Our key insight is that the inference of such a predicate ϕ_i is precisely the familiar classification problem in machine learning. Hence, to find predicate ϕ_i, we first extract relevant features from each path and then use decision tree learning.

2.5.5.1 Feature Extraction

To use decision tree learning for classification, we need to represent each input path using a finite set of discrete features. In the HADES system, these features are domain-specific and therefore defined separately for each application domain. For instance, *some* of the features for the file system domain include file types, permissions, and the presence of a certain file or directory in the path. Given path p, we write $\alpha(p)$ to denote the feature vector for p and $\alpha_f(p)$ for the value of feature f for path p.

2.5.5.2 Decision Tree Learning

We now explain how to use decision tree learning to infer a predicate distinguishing paths Π_1 from those in Π_2. Given sets Π_1 and Π_2 and a set of features $\mathcal{F}$, we use the ID3 algorithm [147] to construct a decision tree $\mathcal{T}_\mathcal{D}$ with the following properties:

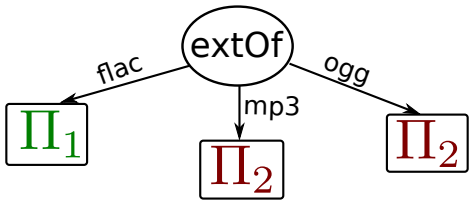
- Each leaf in $\mathcal{T}_\mathcal{D}$ is labeled as Π_1 or Π_2
- Each internal node of $\mathcal{T}_\mathcal{D}$ is labeled with a feature $f \in \mathcal{F}$
- Each edge (f, f', ℓ) from f to f' is annotated with a label ℓ that indicates a possible value of feature f
- Let $(f_1, f_2, \ell_1), \ldots (f_n, \Pi_i, \ell_n)$ be a root-to-leaf path in $\mathcal{T}_\mathcal{D}$. Then, for every $p \in \Pi_1 \cup \Pi_2$, we have:

$$\big(\bigwedge_{i=1}^{n} \alpha_{f_i}(p) = l_i\big) \Leftrightarrow p \in \Pi_i$$

Given such a decision tree $\mathcal{T}_{\mathcal{D}}$, identifying a predicate ϕ differentiating Π_1 from Π_2 is simple. Let $\pi = \langle (f_1, f_2, \ell_1), \ldots (f_n, \Pi_1, \ell_n) \rangle$ be a path in $\mathcal{T}_{\mathcal{D}}$, and let $\varphi(\pi)$ denote the formula $\wedge_{i=1}^{n}(f_i = l_i)$. Assuming Π_1 corresponds to $inputs(\mathcal{E}_i)$ and Π_2 is $inputs(\mathcal{E}) - inputs(\mathcal{E}_i)$, the following DNF formula ϕ_i gives us a classifier for $\mathcal{P}_i$:

$$\phi_i : \bigvee_{\pi \in pathTo(\mathcal{T}_{\mathcal{D}}, \Pi_1)} \varphi(\pi)$$

Example 2.10. *Consider partition $\mathcal{P}_2$ from the example of Section 2.2. Here, $\Pi_1 = \{p_1, p_3\}$ and $\Pi_2 = \{p_2, p_4\}$. After running ID3, we obtain the following decision tree:*



Hence, we extract the classifier $\phi_2 : \text{ExtOf(x)} = \text{flac}$.

2.6 Implementation

We have implemented our synthesis algorithm in a tool called HADES, which consists of $\approx 9,500$ lines of C++ code.The only external tool used by HADES is the Z3 SMT solver [45]. The core of HADES is the domain-agnostic synthesis backend, which accepts input-output examples in the form of hierarchical data trees and emits path transformation functions in the intermediate language of Figure 2.6.

HADES provides an interface for domain-specific plug-ins, and our current implementation incorporates two such front ends: one for XML transformations using XSLT and another one for bash scripts. However, HADES can be extended to new domains by implementing plug-ins that implement the following functionality: (a) represent input-output examples as HDTs; (b) use the synthesized path transformer to emit tree transformation code in the target language; (c) specify any domain-specific functions and features.

2.7 Evaluation

We evaluated HADES by using it to automate 36 data transformation tasks in the file system and XML domains. Our examples come from two sources: on-line forums (e.g., Stackoverflow, bashscript.org) and teaching assistants at our institution. To simulate a real-word usage scenario of HADES where end-users provide input-output examples, we performed a user study involving six students, only three of which are CS majors. The students in our study neither had prior knowledge of HADES nor are they familiar with program synthesis research.

Prior to the evaluation, we gave the participants a demo of the system, explained how to provide input/output examples, and how to check whether the generated script is correct. For each benchmark, we provided the users an English description of the task to be performed as well as a set of test cases to assess whether HADES produces the correct result. The participants were asked to come up with a set of examples for each benchmark, and then run the

Benchmarks			Time	Script			User		
		Description	Total (s)	Branches	Segments	LOC	Iterations	Examples	Depth
File System	F1	Categorize .csv files based on their group	0.03	1	2	47	2	2	3
	F2	Make all script files executable	0.01	2	2	51	1	2	2
	F3	Copy all text and bash files to directory temp	0.05	2	3	56	2	4	3
	F4	Append last 3 directory names to file name and delete directories	0.02	1	2	50	2	2	6
	F5	Put files in directories based on modification year/month/day	0.02	1	4	52	1	2	4
	F6	Copy files without extension into the "NoExtension" directory	0.05	2	3	56	2	7	4
	F7	Archive each directory to a tarball with modify month in its name	0.01	1	1	50	1	2	2
	F8	Make files in "DoNotModify" directory read-only	0.12	2	2	83	2	4	3
	F9	Convert .mp3, .wma, and .m4a files to .ogg	0.02	1	1	47	1	3	3
	F10	Change group of text files in "Public" directory to "everyone"	0.06	2	2	77	1	3	2
	F11	Change directory structure	0.03	1	4	52	2	3	6
	F12	Convert .zip archives to tarballs	0.08	2	2	77	1	3	3
	F13	Organize all files based on their extensions	1.85	4	10	148	1	9	3
	F14	Append modification date to the file name	0.01	1	2	48	1	2	3
	F15	Convert pdf files to swf files	0.03	2	2	55	2	2	2
	F16	Delete files which are not modified last month	0.03	2	2	54	2	5	2
	F17	Convert video files to audio files and put them in "Audio" directory	0.01	1	2	49	1	3	5
	F18	Append "lgst" to name of largest file and "sml" to xml files $\leq$ .1kB	17.94	3	5	110	2	6	3

Table 2.1: File System and XML Benchmarks (part 1)

Benchmarks			Time	Script			User		
		Description	Total (s)	Branches	Segments	LOC	Iterations	Examples	Depth
File System	F19	Extract tarballs to a directory named using file and parent directory	0.36	2	3	83	2	3	3
	F20	Convert xml files $\geq$ 1kB to text files	0.09	2	2	77	2	4	2
	F21	Append parent name to each .c file and copy under "MOSS"	0.04	2	3	56	2	4	4
	F22	Keep all files older than 5 days	0.59	2	2	54	3	7	2
	F23	Copy each file to the directory created with its file name	0.02	1	2	47	1	3	5
	F24	Archive directories which are not older than a week	0.11	2	2	80	3	5	2
XML	X1	Add style='bold' attribute to parent element of each text	0.02	1	2	145	2	3	5
	X2	Merge elements with "status" attribute and put under children	0.02	1	3	169	3	3	5
	X3	Remove all attributes	0.01	1	1	115	2	3	3
	X4	Change the root element of xml	0.02	1	2	222	1	2	3
	X5	Remove 3rd element and put all nested elements under parent	0.02	1	2	129	1	3	4
	X6	Create a table which maps each text to its parent element tag	0.09	2	10	497	1	6	4
	X7	Remove text in element "done" and put all other text under "todo"	0.05	2	3	241	4	8	3
	X8	Generate HTML drop down list from a XML list storing the data	0.02	1	4	443	1	3	3
	X9	Rename a set of element tags to standard HTML tags	0.02	1	4	356	1	2	3
	X10	Move "class" attribute and categorize based on "class"	0.18	2	6	229	1	6	3
	X11	Categorize based on "tag" and put each class under element with valid HTML tag	5.55	3	9	517	2	6	3
	X12	Delete all elements with tag "p"	0.05	2	1	122	1	5	4

Table 2.2: File System and XML Benchmarks (part 2)

tests to verify whether HADES' output is correct. If HADES failed to produce the correct result, the users were asked to restart the process with a modified set of input-output examples.

Tables 2.1 and 2.2 summarizes the results of our evaluation. The column labeled "Description" provides a brief summary of each benchmark, and "Time" reports synthesis time in seconds. The column labeled "Script" gives statistics about the synthesized script, while the column named "User" provides important data related to user interaction.

2.7.1 Performance

To evaluate performance, we utilized the examples provided by the participants from our user study. [7] All performance experiments are conducted on a MacBook Pro with 2.6 GHz Intel Core i5 processor and 8 GB of 1600 MHz DDR3 memory running OS X version 10.10.3.

The column labeled "Time" in Tables 2.1 and 2.2 reports the *total* synthesis time in seconds, including conversion of examples to HDTs and emission of bash or XSLT code. On average, HADES takes 0.90 seconds to synthesize a directory transformation and 0.51 seconds to synthesize an XML transformation. Across all benchmarks, HADES is able to synthesize 91.6% of the benchmarks in under 1 second and 97.2% of the benchmarks in under 10 seconds.

[7] When there were multiple rounds of interaction with the user, we used the examples from the last round.

2.7.2 Complexity

The column labeled "Script" in Tables 2.1 and 2.2 reports various statistics about the script synthesized by HADES: "Branches" reports the number of branches in the synthesized program. ,"Segments" reports the number of loops , and "LOC" gives the number of lines of code for the synthesized *path transformer*. Note that the whole script synthesized by HADES is actually significantly larger (due to furcation/splicing code); the statistics here only include the path transformation portion. As summarized by this data, the scripts synthesized by HADES are fairly complex: They contain between 1-4 branches, 1-10 loops, and between 47-517 lines of code for the path transformer. Furthermore, since our algorithm always generates a simplest path transformer, the reported statistics give a lower bound on the complexity of the required path transformations.

2.7.3 Usability

The last part of Tables 2.1 and 2.2 reports data about our user study: The column "Iteration" reports the number of rounds of tool-user interaction, "Examples" gives the number of furcated examples, and "Depth" indicates the maximum depth of the input-output trees. Our results demonstrate that HADES is user-friendly: 88.8% of the benchmarks require only 1-2 rounds of user interaction, with no task requiring more than 4 rounds. Furthermore, 72.2% of the tasks require less than 5 examples, and no task requires more than 9. Finally, tree depth is typically very small – by providing example trees

with depth 3.2 on average, users are able to obtain scripts that work on trees of unbounded depth.

2.7.4 Comparison with Other Tools

To substantiate our claim that our approach broadens the scope of tree transformations that can be automatically synthesized, we also compared HADES with two existing tools, namely λ^2 [62] and Myth [136], for synthesizing higher-order functional programs. Since these tools do not directly handle Unix directories or XML documents, we manually created a simplified tree abstraction for each task and supplied these tools with a suitable set of input-output examples. Myth was unable to synthesize any of our benchmarks, and λ^2 was only able to synthesize a single example (X3) within a time limit of 600 seconds. Since these tools target a much broader class of synthesis tasks, their search space seems to blow up when presented with non-trivial tree-transformation tasks. In contrast, by learning path transformers that are applied to each path in the tree, our synthesis algorithm can synthesize complex transformations in a practical manner.

2.8 Summary

In this chapter, we presented an algorithm for synthesizing tree transformations from examples. The central idea of our approach is to reduce the generation of tree transformations to the synthesis of list (path) transformations. The path transformations are synthesized using a novel combination

of decision tree learning and SMT solving. The reduction from tree to path transformations simplifies the underlying synthesis algorithm, while allowing us to handle a rich class of tree transformations, including those that restructure the tree.

On the practical side, we have shown that our algorithm has numerous applications for automating the manipulation of hierarchically structured data, such as XML files and Unix directories. In the longer run, approaches like ours can be embedded into end-user programming tools such as Apple's Automator [1], which offers visual abstractions for everyday scripting tasks. Since HADES allows users to generate complex programs from simple examples, it offers a plausible way to broaden the scope of such tools.

Chapter 3

Mitra [1]

While many applications export data in hierarchical formats like XML and JSON, it is often necessary to convert such hierarchical documents to a relational representation. In particular, applications that extensively perform data extraction queries prefer to store and access the data in relational databases to increase their performance. This chapter presents a novel programming-by-example approach, and its implementation in a tool called Mitra, for automatically migrating tree-structured documents to relational tables. We have evaluated the proposed technique using two sets of experiments. In the first experiment, we used Mitra to automate 98 data transformation tasks collected from StackOverflow. Our method can generate the desired program for 94% of these benchmarks with an average synthesis time of 3.8 seconds. In the second experiment, we used Mitra to generate programs that can convert real-world XML and JSON datasets to full-fledged relational databases. Our evaluation shows that Mitra can automate the desired transformation for all datasets.

We start this chapter by motivating our approach in Section 3.1. Then,

[1] Parts of this chapter have appeared in [185].

we illustrate our methodology by describing an example in Section 3.2. Section 3.3 defines a variation of HDTs that we use in Mitra, and Section 3.4 presents Mitra's domain specific language for implementing tree-to-table transformations. We describe the synthesis algorithm in Section 3.5, discuss the implementation details in Section 3.6, and present the experimental results in Section 3.7. Finally, we conclude this chapter with a brief summary in Section 3.8.

3.1 Introduction

3.1.1 Motivation

Many applications store and exchange data using a hierarchical format, such as XML or JSON documents. Such tree-structured data models are a natural fit in cases where the underlying data is hierarchical in nature. Furthermore, since XML and JSON documents incorporate both data and meta-data, they are self-describing and portable. For these reasons, hierarchical data formats are popular for exporting data and transferring them between different applications.

Despite the convenience of hierarchical data models, there are many situations that necessitate converting them to a relational format. This transformation, sometimes referred to as "shredding", may be necessary for a variety of reasons. For example, data stored in an XML document may need to be queried by an existing application that interacts with a relational database. Furthermore, because hierarchical data models are often not well-suited for

efficient data extraction, converting them to a relational format is desirable when query performance is important.

In this chapter, we introduce a new technique based on *programming-by-example (PBE)* [117, 75] for converting hierarchically structured data to a relational format. In our methodology, the user provides a set of simple input-output examples to illustrate the desired transformation, and our system, MITRA [2], automatically synthesizes a program that performs the desired task. Because MITRA learns the target transformation from small input-output examples, it can achieve automation with little guidance from the user. In a typical usage scenario, a user would "train" our system on a small, but representative subset of the input data and then use the program generated by MITRA to convert a very large document to the desired relational representation.

While programming-by-example has been an active research topic in recent years [73, 26, 161, 61, 184, 59, 89, 145], most techniques in this space focus on transformations between similarly structured data, such as string-to-string [73, 161], tree-to-tree [184, 61] or table-to-table transformations [59, 176, 89, 145]. Unfortunately, automating transformations from tree- to table-structured data brings new technical challenges that are not addressed by prior techniques. First, because the source and target data representations are quite different, the required transformations are typically more complex than those between similarly-structured data. Second, since each row in the target table

[2]stands for Migrating Information from Trees to RelAtions

corresponds to a relation between nodes in the input tree, the synthesizer needs to discover these "hidden links" between tree nodes.

3.1.2 Methodology

This chapter addresses these challenges by presenting a new program synthesis algorithm that decomposes the synthesis task into two simpler subproblems that aim to learn the column and row construction logic separately:

- *Learning the column extraction logic:* Given an attribute in a relational table, our approach first synthesizes a program to extract tree nodes that correspond to that attribute. In other words, we first ignore relationships between different tree nodes and construct each column separately. Taking the cross product of the extracted columns yields a table that overapproximates the target table (i.e., contains extra tuples).
- *Learning the row extraction logic:* Since the program learned in the first phase produces a table that overapproximates the target relation, the next phase of our algorithm synthesizes a program that filters out "spurious" tuples generated in the first phase. In essence, the second phase of the algorithm discovers the "hidden links" between different nodes in the original tree structure.

Figure 3.1 shows a schematic illustration of our synthesis algorithm. Given an input tree T and output table R with k columns, our technique first learns k different programs $\pi_1, \ldots, \pi_k$, where each *column extraction*

program π_i extracts from T the data stored in the i'th column of R. Our synthesis algorithm then constructs an intermediate table by applying each π_i to the input tree and taking their cross product. Thus, the intermediate table $\pi_1(T) \times \ldots \times \pi_k(T)$ generated during the first phase overapproximates the target table (i.e.,it may contain more tuples than R). In the next phase, our technique learns a *predicate* ϕ that can be used to filter out exactly the spurious tuples from the intermediate table. Hence, the program synthesized by our algorithm is always of the form $\lambda x.\mathit{filter}(\pi_1 \times \ldots \times \pi_k, \phi)$. Furthermore, since the synthesized program should not be over-fitted to the user-provided examples, our method always learns the *simplest* program of this shape that is consistent with the user-provided input-output examples.

From a technical point of view, our contributions are three-fold. First, we propose a domain-specific language (DSL) that is convenient for expressing transformations between tree-structured and relational data. Our DSL is expressive enough to capture many real-world data transformation tasks, and it also facilitates efficient synthesis by allowing us to decompose the problem into two simpler learning tasks. While the programs in this DSL may sometimes be inefficient, our method eliminates redundancies by memoizing shared computations in the final synthesized program. This strategy allows us to achieve a good trade-off between expressiveness, ease of synthesis, and efficiency of the generated programs.

The second technical contribution is a technique for automatically learning column extraction programs using *deterministic finite automata (DFA)*.

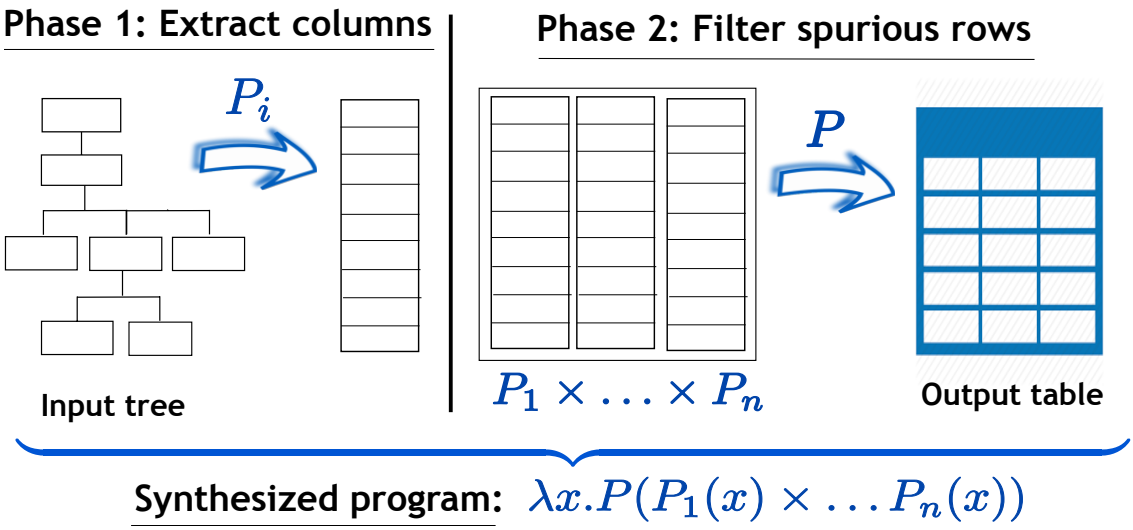


Figure 3.1: Schematic illustration of MITRA's approach

Given an input tree and a column from the output table, our method constructs a DFA whose language corresponds to the set of DSL programs that are consistent with this example. Hence, in our methodology, learning column extraction programs boils down to finding a word (i.e., sequence of DSL operators) that is accepted by the automaton.

Our third technical contribution is a novel technique for learning predicates that can be used to filter out spurious tuples in the intermediate table. Given a set of positive examples (i.e., tuples in the output table) and a set of negative examples (i.e., spurious tuples in the intermediate table), we need to find a smallest classifier in the DSL that can be used to distinguish the positive and negative examples. Our key observation is that this task can be reduced to *integer linear programming*. In particular, our method first generates the

universe of all possible atomic predicates over the DSL and then infers (using integer linear programming) a *smallest* subset of predicates that can be used to distinguish the positive and negative examples. Given such a subset, our method then uses standard logic minimization techniques to find a boolean combination of atomic predicates that can serve as a suitable classifier.

We have implemented our technique in a tool called Mitra, which consists of a language-agnostic synthesis core for tree-to-table transformations as well as domain-specific plug-ins. While Mitra can generate code for migrating data from any tree-structured representation to a relational table, it requires plug-ins to translate the input format into our intermediate representation. Our current implementations contains two such plug-ins for XML and JSON documents. Furthermore, Mitra can be used to transform tree-structured documents to a full-fledged relational database by invoking it once for each table in the target database.

We have evaluated Mitra by performing two sets of experiments. In our first experiment, we use Mitra to automate 98 data transformation tasks collected from StackOverflow. Mitra can successfully synthesize the desired program for 94% of these benchmarks, with an average synthesis time of 3.8 seconds. In our second experiment, we use Mitra to migrate four real-world XML and JSON datasets (namely, IMDB, YELP, MONDIAL, and DBLP) to a full-fledged relational database. Our experiments show that Mitra can perform the desired task for all four datasets.

3.2 Overview

In this section, we give a high-level overview of our technique with the aid of a simple motivating example. Consider the XML file from Figure 3.2a which contains information about the users of a social network as well as the "friendship" relation between them. Suppose a user wants to convert this XML document into the relational table shown in Figure 3.2b. Observe that this transformation is non-trivial because the XML file stores this information as a mapping from each user to a list of friends, where each friend is represented by their `fid`. In contrast, the desired table stores this information as tuples (A, B, n), indicating that person with name A is friends with user with name B for n years.

Suppose that the original XML file is much bigger than the one shown in Figure 3.2a, so the user decides to automate this task by providing the input-output example from Figure 3.2 and uses Mitra to automatically synthesize the desired data migration program. We now give a brief overview of how Mitra generates the target program.

Internally, Mitra represents the input XML file as a *hierarchical data tree*, shown in Figure 3.3. Here, each node corresponds to an element in the XML document, and an edge from n to n' indicates that element n' is inside n. The bold text in each node from Figure 3.3 corresponds to the data stored in that element.

As mentioned in Section 3.1, Mitra starts by learning all possible

```
<Users>
  <Person id="1" name="Alice">
    <Friendship>
      <Friend fid="2" years="3"></Friend>
      <Friend fid="4" years="4"></Friend>
    </Friendship>
  </Person>
  <Person id="2" name="Bob">
    <Friendship>
      <Friend fid="3" years="2"></Friend>
      <Friend fid="1" years="3"></Friend>
    </Friendship>
  </Person>
  <Person id="3" name="Clair">
    <Friendship>
      <Friend fid="2" years="2"></Friend>
    </Friendship>
  </Person>
  <Person id="4" name="David">
    <Friendship>
      <Friend fid="1" years="4"></Friend>
    </Friendship>
  </Person>
</Users>
```

(a) Input XML

Person	Friend-with	Years
Alice	Bob	3
Alice	David	4
Bob	Clair	2
Bob	Alice	3
Clair	Bob	2
David	Alice	4

(b) Output relation

Figure 3.2: MITRA's motivating example

column extraction programs that can be used to obtain column i in the output table from the input tree. Figure 3.5 shows the extraction programs for each column. Specifically, MITRA learns a single column extraction program π_{11} (resp. π_{21}) for the column `Person` (resp. column `Friend-with`). For instance, the program π_{11} first retrieves all children with tag `Person` of the root node, and, for each of these children, it returns the child with tag `name`. Since there are several ways to obtain the data in the `years` column, MITRA learns four different column extractors (namely, $\pi_{31}, \ldots, \pi_{34}$) for `years`.

Next, MITRA conceptually generates intermediate tables by applying

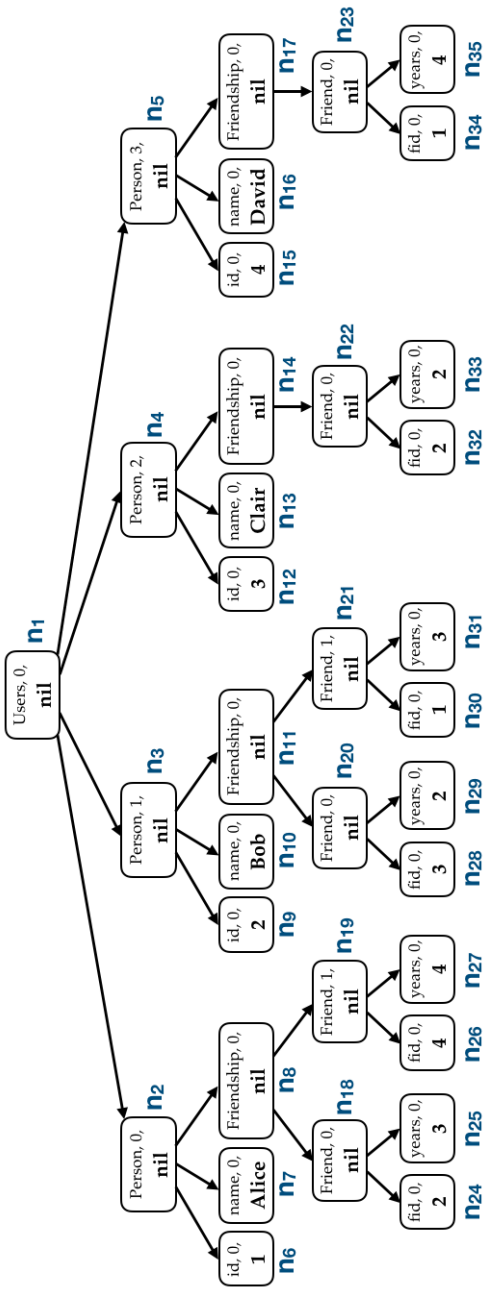


Figure 3.3: *Hierarchical data tree* representation of the input XML

$\pi_{11}(\{n_1\})$	$\pi_{21}(\{n_1\})$	$\pi_{31}(\{n_1\})$
n_7	n_7	n_{25}
n_7	n_7	n_{27}
n_7	n_7	n_{29}
n_7	n_7	n_{31}
n_7	n_7	n_{33}
n_7	n_7	n_{35}
n_7	n_{10}	n_{25}
n_7	n_{10}	n_{27}
...		
n_{16}	n_{16}	n_{31}
n_{16}	n_{16}	n_{33}
n_{16}	n_{16}	n_{35}

Figure 3.4: Intermediate table generated by MITRA

each column extractor to the input tree and taking their cross product. [3] Since we have four different column extraction programs for the `years` attribute, MITRA considers four different intermediate tables, one of which is shown in Figure 3.4. In particular, this table is obtained using the *table extraction* program ψ presented in Figure 3.5. Observe that entries in the intermediate tables generated by MITRA refer to nodes from the input tree.

In the second phase of the synthesis algorithm, MITRA filters out spurious tuples in the intermediate table by learning a suitable *predicate*. For instance, the intermediate table from Figure 3.4 contains several tuples that do

[3]Since this strategy may be inefficient, our implementation performs an optimization to eliminate the generation of intermediate tables in most cases. However, decomposing the problem in this manner greatly facilitates the synthesis task.

$\pi_{11} : pchildren(children(s, Person), name, 0)$
$\pi_{21} : pchildren(children(s, Person), name, 0)$
$\pi_{31} : pchildren(children(\pi_F, Friend), years, 0)$
$\pi_{32} : pchildren(children(\pi_F, Friend), fid, 0)$
$\pi_{33} : pchildren(pchildren(\pi_F, Friend, 0), years, 0)$
$\pi_{34} : pchildren(children(s, Person), id, 0)$
$\pi_F : pchildren(children(s, Person), Friendship, 0)$

$\psi := (\lambda s.\pi_{11})\{root(\tau)\} \times (\lambda s.\pi_{21})\{root(\tau)\} \times (\lambda s.\pi_{31})\{root(\tau)\}$

$P := \lambda\tau.\ filter(\psi, \lambda t.\ \phi_1 \wedge \phi_2)$

$\phi_1 :\ ((\lambda n.\ parent(n))\ t[0]) = ((\lambda n.\ parent(parent(parent(n)))\ t[2])$

$\phi_2 :\ ((\lambda n.\ child(parent(n), id, 0)))\ t[1]) = ((\lambda n.\ child(parent(n), fid, 0))\ t[2])$

Figure 3.5: Synthesized program by MITRA for the motivating example

not appear in the output table. As an example, consider the tuple (n_7, n_7, n_{25}). If we extract the data stored at these nodes, we would obtain the tuple `(Alice, Alice, 3)`, which is not part of the desired output table. To filter out such spurious tuples from the intermediate table, MITRA learns a predicate ϕ such that ϕ evaluates to *true* for every tuple in the target table, and evaluates to $false$ for the spurious tuples. For our running example and the intermediate table from Figure 3.4, MITRA learns the predicate $\phi_1 \wedge \phi_2$, where ϕ_1 and ϕ_2 are shown in Figure 3.5. Here, ϕ_1 ensures that a tuple (a, b, c) only exists if a and c are associated with the same person. Similarly, ϕ_2 guarantees that b refers to the person who has been friends with a for c years. For instance, since ϕ_2 evaluates to false for the first tuple in the intermediate table from Figure 3.4,

this spurious tuple will be filtered out by the learnt predicate.

While all programs synthesized by Mitra are guaranteed to satisfy the input-output examples, not all of these programs may correspond to the user's intent. In particular, since examples are typically under-constrained, there may be multiple programs that satisfy the provided examples. For instance, in our running example, Mitra finds four different programs that are consistent with the examples. Our synthesis algorithm uses a heuristic ranking function based on *Occam's razor* principle to rank-order the synthesized programs. For instance, programs that involve complex predicates are assigned a lower score by the ranking algorithm. Since the program P shown in Figure 3.5 has the highest rank, Mitra chooses it as the desired solution and optimizes P by memoizing redundant computations and avoiding the generation of intermediate tables whenever possible. Finally, Mitra generates an executable XSLT program, which is available from `goo.gl/rcAHT4`. The user can now apply the generated code to the original (much larger) XML document and obtain the desired relational representation. For instance, the synthesized program can be used to migrate an XML document with more than 1 million elements to the desired relational table in 154 seconds.

3.3 A Variant of HDTs

In this section, we introduce a variant of hierarchical data trees which is slightly different from the definition of HDTs presented in Section 2.3.

Definition 3.1. (A variant of) Hierarchical Data Tree *A hierarchical data tree (HDT) τ is a rooted tree represented as a tuple $\langle V, E\rangle$ where V is a set of nodes, and E is a set of directed edges. A node $v \in V$ is a triple $(tag, pos, data)$ where tag is a label associated with node v, pos indicates that v is the pos'th element with label tag under v's parent, and data corresponds to the data stored at node v.*

While Definition 3.1 of HDTs is inspired by Definition 2.1 presented in Section 2.3, there are some subtle differences: In particular, instead of mapping each element to a single node as in Definition 2.1, Definition 3.1 maps each element to multiple tree nodes. Throughout this chapter, HDTs are defined with respect to our new Definition 3.1.

Given a node $n = (t, i, d)$ in a hierarchical data tree, we use the notation $n.tag$, $n.pos$, and $n.data$ to indicate t, i, and d respectively. We also use the notation *isLeaf*(n) to denote that n is a leaf node of the HDT. In the rest of this chapter, we assume that only leaf nodes contain data; hence, for any internal node (t, i, d), we have $d = nil$. Finally, given an HDT τ, we write $\tau.root$ to denote the root node of τ.

3.3.1 XML Documents as HDTs

We can represent XML files as hierarchical data trees where nodes correspond to XML elements. In particular, an edge from node v' to $v = (t, i, d)$ indicates that the XML element e represented by v is nested directly inside element e' represented by v'. Furthermore since v has tag t and position i,

this means e is the i'th child with tag t of e'. We also model XML attributes and text content as nested elements. This design choice allows our model to represent elements that contain a combination of nested elements, attributes, and text content.

Example 3.1. *Figure 3.3 shows the HDT representation of the XML file from Figure 3.2a. Observe how attributes are represented as nested elements in the HDT representation.*

3.3.2 JSON Documents as HDTs

JSON documents store data as a set of nested key-value pairs. We can model JSON files as HDTs in the following way: Each node $n = (t, i, d)$ in the HDT corresponds to a key-value pair e in the JSON file, where t represents the key and d represents the value. Since values in JSON files can be arrays, the position i corresponds to the i'th entry in the array. For instance, if the JSON file maps key k to the array $[18, 45, 32]$, the HDT contains three nodes of the form $(k, 0, 18)$, $(k, 1, 45), (k, 2, 32)$. An edge from n' to n indicates that the key-value pair e represented by n is nested inside the key-value pair e' represented by n'.

Example 3.2. *Figure 3.6 shows the JSON document corresponding to the HDT representation in Figure 3.3. Observe that JSON objects and arrays are represented as internal nodes with data* nil*. For a given node n, we have $n.pos = 0$ unless the parent of n corresponds to an array.*

```
{"Users":
   {"Person":
       [ {"id": 1,  "name": Alice,
          "Friendship": {"Friend": [ {"fid": 2,  "years": 3},
                                     {"fid": 4,  "years": 4}]}
         },
         {"id": 2,  "name": Bob,
          "Friendship": {"Friend": [ {"fid": 3,  "years": 2},
                                     {"fid": 1,  "years": 3}]}
         },
         {"id": 3,  "name": Clair,
          "Friendship": {"Friend": [ {"fid": 2,  "years": 2}]}
         },
         {"id": 4,  "name": David,
          "Friendship": {"Friend": [ {"fid": 1,  "years": 4}]}
         }]
   }
}
```

Figure 3.6: Example of a JSON file

3.4 Domain-Specific Language

In this section, we present our domain-specific language (DSL) for implementing transformations from hierarchical data trees to relational tables. As standard, we represent relational tables as a bag of tuples, where each tuple is represented using a list. Given a relational table $\mathcal{R}$, we use the notation $column(\mathcal{R}, i)$ to denote the i'th column of $\mathcal{R}$, and we use the terms "relation" and "table" interchangably.

Figure 3.7 shows the syntax of our DSL, and Figure 3.8 gives its semantics. Before we explain the constructs in this DSL, an important point is that our language is designed to achieve a good trade-off between expressiveness and efficiency of synthesis. That is, while our DSL can express a large class of tree-to-table transformations that arise in practice, it is designed to make automated synthesis practical.

$$
\begin{array}{rcl}
\text{Program } P & := & \lambda\tau.\mathit{filter}(\psi, \lambda t.\phi) \\
\text{Table Extractor } \psi & := & (\lambda s.\pi)\ \{\mathit{root}(\tau)\} \mid \psi_1 \times \psi_2 \\
\text{Column Extractor } \pi & := & s \mid \mathit{children}(\pi, \mathit{tag}) \\
 & & \mid \mathit{pchildren}(\pi, \mathit{tag}, \mathit{pos}) \mid \mathit{descendants}(\pi, \mathit{tag}) \\
\text{Predicate } \phi & := & \big((\lambda n.\varphi)\ t[i]\big) \trianglelefteq c \mid \big((\lambda n.\varphi_1)\ t[i]\big) \trianglelefteq \big((\lambda n.\varphi_2)\ t[j]\big) \\
 & & \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \\
\text{Node Extractor } \varphi & := & n \mid \mathit{parent}(\varphi) \mid \mathit{child}(\varphi, \mathit{tag}, \mathit{pos})
\end{array}
$$

Figure 3.7: Syntax of MITRA's DSL. Here, τ denotes the input tree, t is bound to a tuple of nodes in τ, and n denotes a node in τ. Furthermore, i and j are integers, and c is a constant value (string, integer, etc). $t[i]$ gives the i-th element in tuple t.

The high-level structure of the DSL follows our synthesis methodology of decomposing the problem into two separate column and row extraction operations. In particular, a program P is of the form $\lambda\tau.\mathit{filter}(\psi, \lambda t.\phi)$, where τ is the input HDT and ψ is a *table extractor* that extracts a relation $\mathcal{R}'$ from τ. As mentioned in Section 3.1, the extracted table $\mathcal{R}'$ overapproximates the target table $\mathcal{R}$. Therefore, the top-level *filter* construct uses a predicate ϕ to filter out tuples in $\mathcal{R}'$ that do not appear in $\mathcal{R}$.

3.4.1 Table Extractor

A *table extractor* ψ constructs a table by taking the cartesian product of a number of columns, where an entry in each column is a "pointer" to a node

$$
\begin{array}{rcl}
[\![\ \mathit{filter}(\psi, \lambda t.\phi)\ ]\!]_{\mathtt{T}} & = & \{\ (\mathtt{n}_1.\mathrm{data}, \cdot\cdot, \mathtt{n}_k.\mathrm{data}) \mid \mathtt{t} \in [\![\psi]\!]_{\mathtt{T}},\ \mathtt{t} = (\mathtt{n}_1, \cdot\cdot, \mathtt{n}_k),\ [\![\phi]\!]_{\mathtt{t},\mathtt{T}} = \mathrm{True}\ \} \\
[\![\ \psi_1 \times \psi_2\ ]\!]_{\mathtt{T}} & = & \{\ (\mathtt{n}_1, \cdot\cdot, \mathtt{n}_k, \mathtt{n}'_1, \cdot\cdot, \mathtt{n}'_l) \mid (\mathtt{n}_1, \cdot\cdot, \mathtt{n}_k) \in [\![\psi_1]\!]_{\mathtt{T}},\ (\mathtt{n}'_1, \cdot\cdot, \mathtt{n}'_l) \in [\![\psi_2]\!]_{\mathtt{T}}\ \} \\
[\![\ (\lambda s.\pi)\ \{\mathrm{root}(\tau)\}\ ]\!]_{\mathtt{T}} & = & \{\ \mathtt{n} \mid\ \mathtt{n} \in [\![\pi]\!]_{\mathtt{s},\mathtt{T}}, \mathtt{s} = \{\mathtt{T}.\mathrm{root}\}\ \} \\
\\
[\![\ x\ ]\!]_{\mathtt{s},\mathtt{T}} & = & \mathtt{s} \qquad \text{where } \mathtt{s} \text{ is a set of nodes in } \mathtt{T} \\
[\![\ \mathit{children}(\pi, tag)\ ]\!]_{\mathtt{s},\mathtt{T}} & = & \{\ \mathtt{n}' \mid \mathtt{n} \in [\![\pi]\!]_{\mathtt{s},\mathtt{T}},\ (\mathtt{n}, \mathtt{n}') \in \mathtt{E},\ \mathtt{n}'.\mathrm{tag} = tag\ \} \\
[\![\ \mathit{pchildren}(\pi, tag, pos)\ ]\!]_{\mathtt{s},\mathtt{T}} & = & \{\ \mathtt{n}' \mid \mathtt{n} \in [\![\pi]\!]_{\mathtt{s},\mathtt{T}},\ (\mathtt{n}, \mathtt{n}') \in \mathtt{E},\ \mathtt{n}'.\mathrm{tag} = tag,\ \mathtt{n}'.\mathrm{pos} = pos\ \} \\
[\![\ \mathit{descendants}(\pi, tag)\ ]\!]_{\mathtt{s},\mathtt{T}} & = & \{\ \mathtt{n}_z \mid \mathtt{n}_1 \in [\![\pi]\!]_{\mathtt{s},\mathtt{T}},\ \exists\{\mathtt{n}_2, ..., \mathtt{n}_{z-1}\} \subset \mathtt{V} \\
 & & \qquad\qquad \text{s.t. } \forall 1 \leq x < z.\ (\mathtt{n}_x, \mathtt{n}_{x+1}) \in \mathtt{E},\ \mathtt{n}_z.\mathrm{tag} = tag\ \} \\
\\
[\![\ \big((\lambda n.\varphi)\ t[i]\big) \trianglelefteq c\ ]\!]_{\mathtt{t},\mathtt{T}} & = & \mathtt{n}'.\mathrm{data} \trianglelefteq c \qquad \text{where } \mathtt{n}' = [\![\varphi]\!]_{\mathtt{n}_i,\mathtt{T}} \\
\\
[\![\ \big((\lambda n.\varphi_1)\ t[i]\big) \trianglelefteq \big((\lambda n.\varphi_2)\ t[j]\big)\ ]\!]_{\mathtt{t},\mathtt{T}} & = & \begin{cases} \mathtt{n}'_1.\mathrm{data} \trianglelefteq \mathtt{n}'_2.\mathrm{data} & \text{if } \mathtt{n}'_1 \text{ and } \mathtt{n}'_2 \text{ are both leaf nodes of } \mathtt{T} \\ \mathtt{n}'_1 = \mathtt{n}'_2 & \text{if } \trianglelefteq \text{ is ``} = \text{''}, \text{ and neither } \mathtt{n}'_1 \text{ nor } \mathtt{n}'_2 \\ & \qquad\qquad \text{is a leaf node of } \mathtt{T} \\ \mathrm{False} & \text{otherwise} \end{cases} \\
 & & \text{where } \mathtt{n}'_1 = [\![\varphi_1]\!]_{\mathtt{n}_i,\mathtt{T}} \text{ and } \mathtt{n}'_2 = [\![\varphi_2]\!]_{\mathtt{n}_j,\mathtt{T}} \\
\\
[\![\ \phi_1 \wedge \phi_2\ ]\!]_{\mathtt{t},\mathtt{T}} & = & [\![\phi_1]\!]_{\mathtt{t},\mathtt{T}}\ \wedge\ [\![\phi_2]\!]_{\mathtt{t},\mathtt{T}} \\
[\![\ \phi_1 \vee \phi_2\ ]\!]_{\mathtt{t},\mathtt{T}} & = & [\![\phi_1]\!]_{\mathtt{t},\mathtt{T}}\ \vee\ [\![\phi_2]\!]_{\mathtt{t},\mathtt{T}} \\
[\![\ \neg\phi\ ]\!]_{\mathtt{t},\mathtt{T}} & = & \neg[\![\phi]\!]_{\mathtt{t},\mathtt{T}} \\
\\
[\![\ n\ ]\!]_{\mathtt{n},\mathtt{T}} & = & \mathtt{n} \\
\\
[\![\ \mathit{parent}(\varphi)\ ]\!]_{\mathtt{n},\mathtt{T}} & = & \begin{cases} \mathtt{n}' & \text{if } (\mathtt{n}', [\![\varphi]\!]_{\mathtt{n},\mathtt{T}}) \in \mathtt{E} \\ \bot & \text{otherwise} \end{cases} \\
\\
[\![\ \mathit{child}(\varphi, tag, pos)\ ]\!]_{\mathtt{n},\mathtt{T}} & = & \begin{cases} \mathtt{n}' & \text{if } ([\![\varphi]\!]_{\mathtt{n},\mathtt{T}}, \mathtt{n}') \in \mathtt{E} \text{ and } \mathtt{n}'.\mathrm{tag} = tag \text{ and } \mathtt{n}'.\mathrm{pos} = pos \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3.8: Semantics of MITRA's DSL. Here, $\mathtt{T} = (\mathtt{V}, \mathtt{E})$ and $\mathtt{t} = (\mathtt{n}_1, \cdot\cdot, \mathtt{n}_l)$.

in the input HDT. Each column is obtained by applying a *column extractor* π to the root node of the input tree. A column extractor π takes as input a set of nodes and an HDT, and returns a set of HDT nodes. Column extractors are defined recursively and can be nested inside each other: The base case simply returns the input set of nodes, and the recursive case extracts other nodes from each input node using the *children*, *pchildren*, and *descendants* constructs. Specifically, the *children* construct extracts all children with a given tag, whereas *pchildren* yields all children with a given tag and specified position. In contrast, the *descendants* construct returns all descendants with the given tag. The formal (denotational) semantics of each construct is given in Figure 3.8.

3.4.2 Predicate

Let us now turn our attention to predicates ϕ that can be used in the top-level *filter* construct. Atomic predicates without boolean connectives are obtained by comparing the data stored in an HDT node with a constant c or the data stored in another tree node. In particular, predicates make use of *node extractors* φ that take a tree node as input and return another tree node. Similar to column extractors, node extractors are recursively defined and can be nested within each other. Observe that node extractors allow accessing both parent and children nodes; hence, they can be used to extract any arbitrary node within the HDT from a given input node. (Figure 3.8 gives the formal semantics).

```
<Dependency>
  <Object id="8"> A
    <Object id="13"> B
      <Object id="18"> D
        <Object id="23"> F </Object>
        <Object id="28"> G </Object>
      </Object>
    </Object>
    <Object id="20"> C
      <Object id="24"> E </Object>
    </Object>
  </Object>
</Dependency>
```

(a) Input XML

Parent	Child
A	B
A	C
B	D
D	F
D	G

(b) Output relation

Figure 3.9: Input-output example for Example 3.3

Going back to the definition of predicates, ϕ takes a tuple t and evaluates to a boolean value indicating whether t should be kept in the output table. The simplest predicate $\big((\lambda n.\varphi)\ t[i]\big) \trianglelefteq c$ first extracts the i'th entry $\mathtt{n}$ of t and then uses the node extractor φ to obtain another tree node $\mathtt{n}'$ from $\mathtt{n}$. This predicate evaluates to true iff $\mathtt{n}'.data \trianglelefteq c$ is true. The semantics of $\big((\lambda n.\varphi_1)\ t[i]\big) \trianglelefteq \big((\lambda n.\varphi_2)\ t[j]\big)$ is similar, except that it compares values stored at two tree nodes $\mathtt{n}, \mathtt{n}'$. In particular, if $\mathtt{n}, \mathtt{n}'$ are both leaf nodes, then we check whether the relation $\mathtt{n}.\text{data} \trianglelefteq \mathtt{n}'.\text{data}$ is satisfied. If they are internal nodes and the operator $\trianglelefteq$ is equality, then we check whether $\mathtt{n}, \mathtt{n}'$ refer to the same node. Otherwise, the predicate evaluates to false. More complex predicates are obtained using the standard boolean connectives $\wedge, \vee, \neg$.

Example 3.3. *Consider the data transformation task illustrated in Figure 3.9, in which we want to map the text value of each object element with id less than* 20 *in the XML file to the text value of its immediate nested object*

$$P := \lambda\tau.\ \mathit{filter}(\psi, \lambda t.\ \phi_1 \wedge \phi_2)$$

$$\psi := (\lambda s.\pi_1)\{\mathit{root}(\tau)\} \times (\lambda s.\pi_2)\{\mathit{root}(\tau)\}$$

$$\pi_1 = \pi_2 = \mathit{pchildren}(\mathit{descendants}(s, \mathit{Object}), \mathit{text}, 0)$$

$$\phi_1 : ((\lambda n.\ \mathit{child}(\mathit{parent}(n), \mathit{id}, 0)))\ t[0]) < 20$$
$$\phi_2 : ((\lambda n.\ \mathit{parent}(n))\ t[0]) = ((\lambda n.\ \mathit{parent}(\mathit{parent}(n))\ t[1])$$

Figure 3.10: Synthesized program for Example 3

elements. Figure 3.10 shows the DSL program that can be used to perform this transformation. Here, column extractors π_1, π_2 *use the* descendants *and* pchildren *constructs to extract all children nodes with tag text and pos* 0 *of any object node reachable from the root. Predicate* ϕ_1 *filters out all tuples where the first element in the tuple has an* id *sibling with value greater than or equal to* 20*. The second predicate* ϕ_2 *ensures that the second element in the tuple is directly nested inside the first one.*

3.5 Synthesis Algorithm

In this section, we present our synthesis algorithm for converting an HDT into a relational table from input-output examples. Our technique can be used to transform an XML or JSON document into a relational database by running the algorithm once for each table in the target database.

The top-level structure of our synthesis algorithm is given in Algorithm 3.1. The algorithm takes a set of input-output examples of the form

Algorithm 3.1 MITRA's top-level synthesis algorithm

1: **procedure** LEARNTRANSFORMATION($\mathcal{E}$)
2: **Input:** Examples $\mathcal{E} = \{\mathtt{T}_1 \rightarrow \mathcal{R}_1, \cdot\cdot, \mathtt{T}_m \rightarrow \mathcal{R}_m\}$.
3: **Output:** Simplest DSL program P^*.
4: **Requires:** Each table $\mathcal{R}_i$ has k columns.
5: **Ensures:** $\forall i \in [1, m] : [\![P^*]\!]_{\mathtt{T}_i} = \mathcal{R}_i$.
6: **for all** $1 \leq j \leq k$ **do**
7: Π_j := LEARNCOLEXTRACTORS($\mathcal{E}$, j);
8: $\Psi := \Pi_1 \times \cdot\cdot \times \Pi_n$;
9: $P^* := \bot$;
10: **for all** $\psi \in \Psi$ **do**
11: ϕ := LEARNPREDICATE($\mathcal{E}$, ψ);
12: **if** $\phi \neq \bot$ **then**
13: $P := \lambda\tau.\mathit{filter}(\psi, \lambda t.\phi)$;
14: **if** $\theta(P) < \theta(P^*)$ **then** $P^* := P$
15: **return** P^*;

$\{\mathtt{T}_1 \rightarrow \mathcal{R}_1, \cdot\cdot, \mathtt{T}_m \rightarrow \mathcal{R}_m\}$, where each $\mathtt{T}_i$ represents an HDT (input example) and $\mathcal{R}_i$ represents its desired relational table representation (output example). Since the schema of the target table is typically fixed in practice, we assume that all output tables (i.e., $\mathcal{R}_i$'s) have the same number of columns. Given these input-output examples, the procedure LEARNTRANSFORMATION returns a program P^* in our DSL that is consistent with all input-output examples. Furthermore, since P^* is the simplest DSL program that satisfies the specification, it is expected to be a general program that is not over-fitted to the examples.

As mentioned earlier, our methodology decomposes the synthesis task

into two phases for extracting columns and rows. In the first phase, we synthesize a *column extraction* program that yields an overapproximation of each column in the output table $\mathcal{R}$. The cartesian product of columns extracted by these column extraction programs further yields a table $\mathcal{R}'$ that overapproximates the target table $\mathcal{R}$. In the second phase, we learn a predicate ϕ that allows us to filter out exactly the "spurious" tuples in $\mathcal{R}'$ that do not appear in the output table $\mathcal{R}$.

Let us now consider the LearnTransformation procedure from Algorithm 3.1 in more detail. Given the examples, we first invoke a procedure called LearnColExtractors that learns a *set* Π_j of column extraction expressions such that applying any $\pi \in \Pi_j$ on each $\mathtt{T}_i$ yields a column that overapproximates the j'th column in table $\mathcal{R}_i$ (lines 6-7). Observe that our algorithm learns a set of column extractors (instead of just one) because some of them might not lead to a desired *filter* program. Our procedure for learning column extractors is based on deterministic finite automata and will be described in more detail in Section 3.5.1.

Once we have the individual column extraction programs for each column, we then obtain the set of all possible table extraction programs by taking the cartesian product of each column extraction program (line 8). Hence, applying each table extraction program $\psi \in \Psi$ to the input tree $\mathtt{T}_i$ yields a table $\mathcal{R}'_i$ that overapproximates the output table $\mathcal{R}_i$.

The next step of the synthesis algorithm (lines 9-14) learns the predicate used in the top-level *filter* construct. For each table extraction program $\psi \in \Psi$,

we try to learn a predicate ϕ such that $\lambda\tau.\mathit{filter}(\psi, \lambda t.\phi)$ is consistent with the examples. Specifically, the procedure LearnPredicate yields a predicate that allows us to filter out spurious tuples in $[\![\psi]\!]_{\mathrm{T}_i}$. If there is no such predicate, LearnPredicate returns $\perp$. Our predicate learning algorithm uses integer linear programming to infer a *simplest* formula with the minimum number of atomic predicates. We describe the LearnPredicate algorithm in more detail in Section 3.5.2.

Since there may be multiple DSL programs that satisfy the provided examples, our method uses the Occam's razor principle to choose between different solutions. In particular, Algorithm 3.1 uses a heuristic cost function θ to determine which solution is simpler (line 14), so it is guaranteed to return a program that minimizes the value of θ. Intuitively, θ assigns a higher cost to programs that use complex predicates or column extractors. We discuss the design of the heuristic cost function θ in Section 3.6.

3.5.1 Learning Column Extraction Programs

Our technique for learning column extraction programs is based on deterministic finite automata (DFA). Given a tree (input example) and a single column (output example), our method constructs a DFA whose language accepts exactly the set of column extraction programs that are consistent with the input-output example. If we have multiple input-output examples, we construct a separate DFA for each example and then take their intersection. The language of the resulting DFA includes column extraction programs that

Algorithm 3.2 Algorithm for learning column extractors

1: **procedure** LEARNCOLEXTRACTORS($\mathcal{E}$, i)
2: **Input:** Examples $\mathcal{E}$ and column number i.
3: **Output:** Set Π of column extractors.
4: **Ensures:** $\forall \pi \in \Pi.\ \forall (\mathtt{T}, \mathcal{R}) \in \mathcal{E}.$
5: $\llbracket \pi \rrbracket_{\{\mathtt{T}.\mathrm{root}\},\mathtt{T}} \supseteq \mathit{column}(\mathcal{R}, i).$
6: $\mathcal{E}' := \{(\mathtt{T}, \kappa) \mid (\mathtt{T}, \mathcal{R}) \in \mathcal{E} \wedge \kappa = \mathit{column}(\mathcal{R}, i)\}$
7: $\mathcal{A} :=$ CONSTRUCTDFA(e_0) where $e_0 \in \mathcal{E}'$
8: **for all** $e \in \mathcal{E}' \backslash \{e_0\}$ **do**
9: $\mathcal{A}' :=$ CONSTRUCTDFA(e)
10: $\mathcal{A} :=$ INTERSECT($\mathcal{A}, \mathcal{A}'$)
11: **return** LANGUAGE($\mathcal{A}$)

are consistent with *all* examples.

Let us now look at the LEARNCOLEXTRACTORS procedure shown in Algorithm 3.2 in more detail. It takes the set of input-output examples $\mathcal{E}$ as well as a column number i for which we would like to learn the extractor. The algorithm returns a set of column extraction programs Π such that for every $\pi \in \Pi$ and every input-output example $(\mathtt{T}, \mathcal{R})$, we have $\llbracket \pi \rrbracket_{\{\mathtt{T}.\mathrm{root}\},\mathtt{T}} \supseteq \mathit{column}(\mathcal{R}, i)$.

Algorithm 3.2 starts by constructing a set of examples $\mathcal{E}'$ mapping each input tree to the i'th column of the output table (line 6). Then, for each example $e \in \mathcal{E}'$, we construct a DFA that represents all column extraction programs consistent with e (lines 7-10). The set of programs consistent with all examples $\mathcal{E}'$ corresponds to the language of the intersection automaton $\mathcal{A}$ from line 11. In particular, the INTERSECT procedure used in line 10 is the

standard intersection procedure for DFAs [86]. Concretely, the intersection of two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ only accepts programs that are accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$ and is constructed by taking the cartesian product of $\mathcal{A}_1$ and $\mathcal{A}_2$ and defining the accepting states to be all states of the form (q_1, q_2) where q_1 and q_2 are accepting states of $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively.

The key part of the LearnColExtractors procedure is the ConstructDFA method, which constructs a deterministic finite automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ from an input-output example using the rules shown in Figure 3.11. Here, the states Q of the automaton correspond to sets of nodes in the input HDT. We use the notation $q_{\texttt{s}}$ to denote the state representing the set of nodes `s`. The alphabet Σ corresponds to the names of column extraction functions in the DSL. Specifally, we have:

$$\begin{aligned}\Sigma &= \{\mathit{children}_{\texttt{tag}} \mid \texttt{tag} \text{ is a tag in } \texttt{T}\} \\ &\cup \{\mathit{pchildren}_{\texttt{tag,pos}} \mid \texttt{tag (pos)} \text{ is a tag (pos) in } \texttt{T}\} \\ &\cup \{\mathit{descendants}_{\texttt{tag}} \mid \texttt{tag} \text{ is a tag in } \texttt{T}\}\end{aligned}$$

In other words, each symbol in the alphabet corresponds to a DSL operator (instantiated with labels and positions from the input HDT). Transitions in the DFA are constructed using the semantics of DSL operators: Intuitively, given a DSL construct $f \in \{\mathit{children}, \mathit{pchildren}, \mathit{descendants}\}$ and a state $q_{\texttt{s}}$, the DFA contains a transition $q_{\texttt{s}} \xrightarrow{f} q'_{\texttt{s}}$ if applying f to `s` produces $\texttt{s}'$. The initial state of the DFA is $\{\texttt{T}.\mathit{root}\}$, and we have $q_{\texttt{s}} \in F$ iff `s` overapproximates the i'th column in table $\mathcal{R}$.

Let us now look at the construction rules shown in Figure 3.11 in more detail. Rules (1)-(4) process the column extractor constructs in our DSL and

$$\frac{\mathtt{s} = \{\mathtt{T}.\mathrm{root}\}}{q_0 = q_{\mathtt{s}} \in Q} \tag{1}$$

$$\frac{\begin{array}{c} q_{\mathtt{s}} \in Q \quad \mathtt{tag} \text{ is a tag in } \mathtt{T} \\ [\![\mathit{children}(s, \mathtt{tag})]\!]_{\mathtt{s},\mathtt{T}} = \mathtt{s}' \end{array}}{q_{\mathtt{s}'} \in Q,\ \delta(q_{\mathtt{s}}, \mathit{children}_{\mathtt{tag}}) = q_{\mathtt{s}'}} \tag{2}$$

$$\frac{\begin{array}{c} q_{\mathtt{s}} \in Q \quad \mathtt{tag} \text{ is a tag in } \mathtt{T} \quad \mathtt{pos} \text{ is a pos in } \mathtt{T} \\ [\![\mathit{pchildren}(s, \mathtt{tag}, \mathtt{pos})]\!]_{\mathtt{s},\mathtt{T}} = \mathtt{s}' \end{array}}{q_{\mathtt{s}'} \in Q,\ \delta(q_{\mathtt{s}}, \mathit{pchildren}_{\mathtt{tag},\mathtt{pos}}) = q_{\mathtt{s}'}} \tag{3}$$

$$\frac{\begin{array}{c} q_{\mathtt{s}} \in Q \quad \mathtt{tag} \text{ is a tag in } \mathtt{T} \\ [\![\mathit{descendants}(s, \mathtt{tag})]\!]_{\mathtt{s},\mathtt{T}} = \mathtt{s}' \end{array}}{q_{\mathtt{s}'} \in Q,\ \delta(q_{\mathtt{s}}, \mathit{descendants}_{\mathtt{tag}}) = q_{\mathtt{s}'}} \tag{4}$$

$$\frac{\mathtt{s} \supseteq \mathit{column}(\mathcal{R}, i)}{q_{\mathtt{s}} \in F} \tag{5}$$

Figure 3.11: DFA construction rules. T is the input tree, $\mathcal{R}$ is the output table, and i is the column to be extracted.

construct states and/or transitions. The first rule adds $q_{\{\mathtt{T}.\mathrm{root}\}}$ as an initial state because the root node of the HDT is directly reachable. The second rule adds a state $q_{\mathtt{s}'}$ and a transition $\delta(q_{\mathtt{s}}, \mathit{children}_{\mathtt{tag}}) = q_{\mathtt{s}'}$ if $\mathit{children}(\mathtt{s}, \mathtt{tag})$ evaluates to $\mathtt{s}'$. Rules (3) and (4) are similar to rule (2) and process the remaining column extraction functions *pchildren* and *descendants*. For example, we have $\delta(q_{\mathtt{s}}, \mathit{pchildren}_{\mathtt{tag},\mathtt{pos}}) = q_{\mathtt{s}'}$ if $\mathit{pchildren}(\mathtt{s}, \mathtt{tag}, \mathtt{pos})$ evaluates to $\mathtt{s}'$. The last rule in Figure 3.11 identifies the final states. In particular, a state $q_{\mathtt{s}}$ is a final state if s is a superset of the target column (i.e., output example).

Theorem 3.1.[4] *Let $\mathcal{A}$ be the DFA constructed by Algorithm 3.2 for a set of input-output examples $\mathcal{E}$ and a column number i. Then, $\mathcal{A}$ accepts a column extraction program π in our DSL iff* $\forall(T, \mathcal{R}) \in \mathcal{E}.\ [\![\pi]\!]_{\{T.root\},T} \supseteq \text{column}(\mathcal{R}, i)$.

Example 3.4. *Suppose the DFA constructed using rules from Figure 3.11 accepts the word ab where* $a = \text{descendants}_{object}$ *and* $b = \text{pchildren}_{text,0}$. *This word corresponds to the following column extractor program:*

$$\pi = \text{pchildren}(\text{descendants}(s, \text{object}), \text{text}, 0)$$

If the input example is T *and the output example is* $\text{column}(\mathcal{R}, i)$, *then we have* $((\lambda s.\pi)\ \{T.root\}) \supseteq \text{column}(\mathcal{R}, i)$.

3.5.2 Learning Predicates

We now turn our attention to the predicate learning algorithm LearnPredicate shown in Algorithm 3.3. This procedure takes the input-output examples $\mathcal{E}$ and a candidate table extractor ψ and returns a predicate ϕ such that for every $(\mathtt{T}, \mathcal{R}) \in \mathcal{E}$, the program $\mathit{filter}(\psi, \lambda t.\phi)$ yields the desired output table $\mathcal{R}$ on input $\mathtt{T}$.

The algorithm starts by constructing a (finite) universe Φ of all possible atomic predicates that can be used in formula ϕ (line 6). These predicates are constructed for a set of input-output examples $\mathcal{E}$ and a candidate table extractor ψ using rules from Figure 3.12. While these rules are not very important for understanding the key ideas of our technique, let us consider rule (4) from

[4]Proofs of all theorems are given in Appendix A.

Algorithm 3.3 Algorithm for learning predicates

1: **procedure** LEARNPREDICATE($\mathcal{E}$, ψ)
2: **Input:** Examples $\mathcal{E}$, a candidate table extractor ψ.
3: **Output:** Desired predicate ϕ.
4: **Requires:** $\forall(\mathtt{T}, \mathcal{R}) \in \mathcal{E}.\ \mathcal{R} \subseteq [\![\psi]\!]_{\mathtt{T}}$.
5: **Ensures:** $\forall(\mathtt{T}, \mathcal{R}) \in \mathcal{E}.\ [\![\mathit{filter}(\psi, \lambda t.\phi)]\!]_{\mathtt{T}} = \mathcal{R}$.
6: $\Phi :=$ CONSTRUCTPREDUNIVERSE($\mathcal{E}$, ψ)
7: $\mathcal{E}^+ := \emptyset;\ \mathcal{E}^- := \emptyset$
8: **for all** $(\mathtt{T}, \mathcal{R}) \in \mathcal{E}$ **do**
9: **for all** $t \in [\![\psi]\!]_{\mathtt{T}}$ **do**
10: **if** $t \in \mathcal{R}$ **then**
11: $\mathcal{E}^+ := \mathcal{E}^+ \cup \{t\}$
12: **else** $\mathcal{E}^- := \mathcal{E}^- \cup \{t\}$
13: $\Phi^* :=$ FINDMINCOVER($\Phi, \mathcal{E}^+, \mathcal{E}^-$)
14: Find ϕ such that:
15: (1) ϕ is boolean combination of preds in Φ^*
16: (2) $\phi(t) = \begin{cases} 1 & \mathit{if}\ \ t \in \mathcal{E}^+ \\ 0 & \mathit{if}\ \ t \in \mathcal{E}^- \end{cases}$
17: **return** ϕ

Figure 3.12 as an example. According to this rule, we generate an atomic predicate $\big((\lambda n.\varphi)\ t[i]\big) \trianglelefteq c$ if i is a valid column number in the range $[1, k]$, c is a constant in one of the input tables, and φ is a "valid" node extractor for column i. Rules (1)-(3) in Figure 3.12 define what it means for a node extractor φ to be valid for column i, denoted as $\varphi \in \chi_i$. In particular, we say φ is a valid node extractor for column i if applying φ does not "throw an exception" (i.e., yield $\perp$) for any of the entries in the i'th column of the generated intermediate

Algorithm 3.4 Algorithm to find minimum predicate set

1: **procedure** FINDMINCOVER(Φ, $\mathcal{E}^+$, $\mathcal{E}^-$)

2: **Input:** Universe of predicates Φ

3: **Input:** Positive examples $\mathcal{E}^+$, negative examples $\mathcal{E}^-$

4: **Output:** Set of predicates Φ^* where $\Phi^* \subseteq \Phi$

5: **for all** $(\phi_k, e_i, e_j) \in \Phi \times \mathcal{E}^+ \times \mathcal{E}^-$ **do**

6: $$a_{ijk} = \begin{cases} 1 & \textit{if } \phi_k(e_i) \neq \phi_k(e_j) \\ 0 & \textit{otherwise} \end{cases}$$

7: $$\textit{minimize} \sum_{k=1}^{|\Phi|} x_k$$

8: *subject to:*

9: $$\forall (e_i, e_j) \in \mathcal{E}^+ \times \mathcal{E}^-.\ \sum_{k=1}^{|\Phi|} a_{ijk} \cdot x_k \geq 1$$

10: $$\wedge\ \forall k \in [1, |\Phi|].\ x_k \in \{0, 1\}$$

11: **return** $\{\phi_i \mid \phi_i \in \Phi \wedge x_i = 1\}$

tables. [5]

The next step of LEARNPREDICATE constructs a set of *positive* and *negative* examples to be used in LEARNPREDICATE (lines 7–12). In this context, positive examples $\mathcal{E}^+$ refer to tuples that should be present in the desired output table, whereas negative examples $\mathcal{E}^-$ correspond to tuples that should be filtered out. The goal of the predicate learner is to find a formula ϕ over atomic predicates in Φ such that ϕ evaluates to true for all positive examples and to false for all negative examples. In other words, formula ϕ

[5] Recall that these intermediate tables are obtained by applying ψ to the input HDTs in $\mathcal{E}$.

$$\frac{}{\pi_i, \mathcal{E} \vdash n \in \chi_i} \tag{1}$$

$$\frac{\begin{array}{c}\pi_i, \mathcal{E} \vdash \varphi \in \chi_i \\ \forall \mathtt{T} \to \mathcal{R} \in \mathcal{E}.\ \forall \mathtt{n} \in \pi_i(\mathtt{T}).\ [\![\mathit{parent}(\varphi)]\!]_{\mathtt{n},\mathtt{T}} \neq \bot\end{array}}{\pi_i, \mathcal{E} \vdash \mathit{parent}(\varphi) \in \chi_i} \tag{2}$$

$$\frac{\begin{array}{c}\pi_i, \mathcal{E} \vdash \varphi \in \chi_i \\ \forall \mathtt{T} \to \mathcal{R} \in \mathcal{E}.\ \forall \mathtt{n} \in \pi_i(\mathtt{T}).\ [\![\mathit{child}(\varphi, \mathtt{tag}, \mathtt{pos})]\!]_{\mathtt{n},\mathtt{T}} \neq \bot\end{array}}{\pi_i, \mathcal{E} \vdash \mathit{child}(\varphi, \mathtt{tag}, \mathtt{pos}) \in \chi_i} \tag{3}$$

$$\frac{\begin{array}{c}\psi = \pi_1 \times \ldots \times \pi_k, \quad i \in [1, k] \\ \pi_i, \mathcal{E} \vdash \varphi \in \chi_i \\ \exists(\mathtt{T}, \mathcal{R}) \in \mathcal{E}.\ c \in \mathit{data}(\mathtt{T})\end{array}}{\psi, \mathcal{E} \vdash \big((\lambda n.\varphi)\ t[i]\big) \trianglelefteq c \in \Phi} \tag{4}$$

$$\frac{\begin{array}{c}\psi = \pi_1 \times \ldots \times \pi_k, \quad i \in [1, k], \quad j \in [1, k] \\ \pi_i, \mathcal{E} \vdash \varphi_1 \in \chi_i \\ \pi_j, \mathcal{E} \vdash \varphi_2 \in \chi_j\end{array}}{\psi, \mathcal{E} \vdash \big((\lambda n.\varphi_1)\ t[i]\big) \trianglelefteq \big((\lambda n.\varphi_2)\ t[j]\big) \in \Phi} \tag{5}$$

Figure 3.12: Predicate universe construction rules. $\mathcal{E}$ is the input-output examples and $\psi = \pi_1 \times ... \times \pi_k$ is the candidate table extractor. Here χ_i indicates a set of node extractors that can be applied to the nodes extracted for column i.

serves as a classifier between $\mathcal{E}^+$ and $\mathcal{E}^-$.

To learn a suitable classifier, our algorithm first learns a *minimum* set of atomic predicates that are necessary for distinguishing the $\mathcal{E}^+$ samples from the $\mathcal{E}^-$ ones. Since our goal is to synthesize a general program that is not

over-fitted to the input-output examples, it is important that the synthesized predicate ϕ is as simple as possible. We formulate the problem of finding a simplest classifier as a combination of *integer linear programming (ILP)* and *logic minimization* [126, 146]. In particular, we use ILP to learn a minimum set of predicates Φ^* that must be used in the classifier, and then use circuit minimization techniques to find a DNF formula ϕ over Φ^* with the minimum number of boolean connectives.

Our method for finding the minimum set of atomic predicates is given in Algorithm 3.4. The FindMinCover procedure takes as input the universe Φ of all predicates as well as positive and negative examples $\mathcal{E}^+$, $\mathcal{E}^-$. It returns a subset Φ^* of Φ such that, for every pair of examples $(e_1, e_2) \in \mathcal{E}^+ \times \mathcal{E}^-$, there exists an atomic predicate $p \in \Phi^*$ such that p evaluates to different truth values for e_1 and e_2 (i.e., p differentiates e_1 and e_2).

We solve this optimization problem by reducing it to 0-1 ILP in the following way: First, we introduce an indicator variable x_k such that $x_k = 1$ if $p_k \in \Phi$ is chosen to be in Φ^* and $x_k = 0$ otherwise. Then, for each predicate p_k and every pair of examples $(e_i, e_j) \in \mathcal{E}^+ \times \mathcal{E}^-$, we introduce a (constant) variable a_{ijk} such that we assign a_{ijk} to 1 if predicate p_k distinguishes e_i and e_j and to 0 otherwise. Observe that the value of each a_{ijk} is known, whereas the assignment to each variable x_k is to be determined.

To find an optimal assignment to variables $\vec{x}$, we set up the 0-1 ILP problem shown in lines 7–10 of Algorithm 3.4. First, we require that for every pair of examples, there exists a predicate p_k that distinguishes them. This

requirement is captured using the constraint in line 9: Since a_{ijk} is 1 iff p_k distinguishes e_i and e_j, the constraint at line 9 is satisfied only when we assign at least one of the x_k's differentiating e_i and e_j to 1. The objective function at line 7 minimizes the sum of the x_k's, thereby forcing us to choose the minimum number of predicates that are sufficient to distinguish every pair of positive and negative examples.

Going back to Algorithm 3.3, the return value Φ^* of FindMinCover (line 13) corresponds to a minimum set of predicates that must be used in the classifier; however, we still need to find a *boolean combination* of predicates in Φ^* that differentiates $\mathcal{E}^+$ samples from the $\mathcal{E}^-$ ones. Furthermore, we would like to find the *smallest* such boolean combination for three reasons: (1) large formulas might hinder the generality of the synthesized program as well as its readability, and (2) large formulas would incur more overhead when being evaluated at runtime.

We cast the problem of finding a smallest classifier over predicates in Φ^* as a *logic minimization* problem [126, 146]. In particular, given a set of predicates Φ^*, our goal is to find a smallest DNF formula ϕ over predicates in Φ^* such that ϕ evaluates to *true* for any positive example and to *false* for any negative example. To solve this problem, we start by constructing a (partial) truth table, where the rows correspond to examples in $\mathcal{E}^+ \cup \mathcal{E}^-$, and the columns correspond to predicates in Φ^*. The entry in the i'th row and j'th column of the truth table is *true* if predicate $p_j \in \Phi^*$ evaluates to true for example e_i and false otherwise. The target boolean function ϕ should evaluate

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7
e_1^+	$true$	$true$	$false$	$false$	$true$	$true$	$false$
e_2^+	$false$	$true$	$true$	$true$	$true$	$false$	$true$
e_3^+	$false$	$true$	$true$	$true$	$false$	$false$	$false$
e_1^-	$false$	$false$	$true$	$true$	$false$	$false$	$false$
e_2^-	$false$	$true$	$true$	$true$	$false$	$false$	$true$
e_3^-	$true$	$false$	$true$	$false$	$false$	$false$	$true$

Figure 3.13: Initial truth table for Example 3.5. Initial truth table for the predicate universe Φ, positive examples $\mathcal{E}^+$ and negative examples $\mathcal{E}^-$.

to true for any $e^+ \in \mathcal{E}^+$ and false for any $e^- \in \mathcal{E}^-$. Since we have a truth table describing the target boolean function, we can use standard techniques, such as the Quine-McCluskey method [126, 146], to find a smallest DNF formula representing classifier ϕ.

Theorem 3.2. *Given examples $\mathcal{E}$ and table extractor ψ such that $\forall(T, \mathcal{R}) \in \mathcal{E}.\ \mathcal{R} \subseteq [\![\psi]\!]_T$, Algorithm 3.3 returns a smallest DNF formula ϕ such that $\forall(T, \mathcal{R}) \in \mathcal{E}.\ [\![\text{filter}(\psi, \lambda t.\phi)]\!]_T = \mathcal{R}$ if such a formula exists in our DSL.*

Example 3.5. *Consider predicate universe $\Phi = \{\phi_1, ..., \phi_7\}$, a set of positive examples $\mathcal{E}^+ = \{e_1^+, e_2^+, e_3^+\}$, and a set of negative examples $\mathcal{E}^- = \{e_1^-, e_2^-, e_3^-\}$ with the truth table given in Figure 3.13. Here, the entry at row e_i and column ϕ_j of Figure 3.13 is* true *iff tuple e_i satisfies predicate ϕ_j. The goal of the predicate learner is to find a formula $\phi_?$ with the minimum number of atomic predicates $\phi_i \in \Phi$ such that it evaluates to true for all positive examples and to false for all negative examples. In order to do so, the* FindMinCover *procedure*

	υ_{11}	υ_{12}	υ_{13}	υ_{21}	υ_{22}	υ_{23}	υ_{31}	υ_{32}	υ_{33}
ϕ_1	1	1	0	0	0	1	0	0	1
ϕ_2	1	0	1	1	0	1	1	0	1
ϕ_3	1	1	1	0	0	0	0	0	0
ϕ_4	1	1	0	0	0	1	0	0	1
ϕ_5	1	1	1	1	1	1	0	0	0
ϕ_6	1	1	1	0	0	0	0	0	0
ϕ_7	0	1	1	1	0	0	0	1	1

Figure 3.14: Values of a_{ijk} assigned in line 6 of Algorithm 3.4. Here υ_{ij} corresponds to $(e_i, e_j) \in \mathcal{E}^+ \times \mathcal{E}^-$.

	ϕ_2	ϕ_5	ϕ_7	$\phi_?$
e_1^+	$true$	$true$	$false$	$true$
e_2^+	$true$	$true$	$true$	$true$
e_3^+	$true$	$false$	$false$	$true$
e_1^-	$false$	$false$	$false$	$false$
e_2^-	$true$	$false$	$true$	$false$
e_3^-	$false$	$false$	$true$	$false$

Figure 3.15: Truth table for Example 3.5, constructed in lines $14-16$ of Algorithm 3.3.

first finds the minimum required subset of atomic predicates Φ^* *as described in Algorithm 3.4. Figure 3.14 shows the values of* a_{ijk} *assigned in line 6 of Algorithm 3.4. In particular, the entry at row* ϕ_i *and column* v_{jk} *of Figure 3.14 is* true *iff* a_{ijk} *is true. The highlighted rows in the Figure 3.14 indicate the predicates which are selected to be in* Φ^* *using integer linear programming. After finding* $\Phi^* = \{\phi_2, \phi_5, \phi_7\}$*, lines 14–16 Algorithm 3.3 generate the (partial) truth table as shown in Figure 3.15. The smallest DNF formula that is consistent*

with this truth table is $\phi_5 \vee (\phi_2 \wedge \neg\phi_7)$*, so our algorithm returns this formula as a classifier.*

3.5.3 Synthesis Algorithm Properties

The following two theorems state the soundness and completeness of our algorithm:

Theorem 3.3. (Soundness) *Given examples* $\mathcal{E} = \{T_1 \to \mathcal{R}_1, \cdot\cdot, T_m \to \mathcal{R}_m\}$*, suppose that* LearnTransformation$(\mathcal{E})$ *yields* P^**. Then,* $\forall(T_i, \mathcal{R}_i) \in \mathcal{E}$*, we have* $[\![P^*]\!]_{T_i} = \mathcal{R}_i$*.*

Theorem 3.4. (Completeness) *Suppose there is a DSL program consistent with examples* $\mathcal{E} = \{T_1 \to \mathcal{R}_1, \cdot\cdot, T_m \to \mathcal{R}_m\}$*. Then,* LearnTransformation$(\mathcal{E})$ *eventually returns a program* P^* *such that* $\forall i \in [1, m] : [\![P^*]\!]_{T_i} = \mathcal{R}_i$*.*

Finally, the following theorem states that our synthesis algorithm returns a simplest DSL program with respect to the cost metric θ:

Theorem 3.5. (Simplicity) *Given examples* $\mathcal{E}$*, Algorithm 3.1 returns a DSL program* P^* *such that for any program* P *satisfying* $\mathcal{E}$*, we have* $\theta(P^*) \leq \theta(P)$*.*

3.5.3.1 Complexity

Our algorithm has worst-case exponential time complexity with respect to the size of input-output examples, as *integer linear programming* and *logic minimization* for a given truth table are both NP-hard problems. However, *in*

practice, the complexity of our algorithm does not come close to the worst-case scenario. The next two paragraphs present a more detailed explanation of the empirical complexity of the proposed algorithm.

Let m be the number of input-output examples, k and r be (respectively) the number of columns and maximum number of rows in output tables, and n be the maximum number of nodes in an input tree. Despite the worst-case exponential time complexity in theory, the empirical complexity of our algorithm is close to $O(m^2 \cdot k^2 \cdot n^2 \cdot r \cdot (\log n)^k)$. To see where this result comes from, we first discuss the complexity of the LearnColExtractor procedure. The complexity of DFA construction is $O(n)$, and the complexity of generating the intersection of two DFAs is $O(n^2)$. Therefore, the overall complexity of Algorithm 3.2 is $O(m \cdot n^2)$, and as the result, the loop in lines $4-5$ of Algorithm 3.1 takes $O(k \cdot m \cdot n^2)$. Our empirical evaluations show that the cartesian-product generated in line 6 of the LearnTransformation procedure usually contains a small number of table extractors.

Now we discuss the complexity of the LearnPredicate procedure for a given table extractor ψ and the set of examples $\mathcal{E}$. It first generates the universe of all atomic predicates which contains $O(k^2 \cdot n^2)$ predicates. Then, it constructs the set of positive and negative examples. The number of positive examples is determined by the given input-output examples, which is $O(m \cdot r)$. The number of negative examples for each $(\mathtt{T}, \mathcal{R}) \in \mathcal{E}$ is bounded by the number of tuple in the corresponding intermediate table ($[\![\psi]\!]_{\mathtt{T}}$). In practice, each column extractor usually extracts $O(\log n)$ nodes from a given

tree with n nodes, therefore the intermediate table contains $O((\log n)^k)$ tuples. Hence, the total size of negative examples is $O(m \cdot (\log n)^k)$. In our experiments, Algorithm 3.4 returned a smallest set of atomic predicates, which included a few predicates, in $O(m^2 \cdot k^2 \cdot n^2 \cdot r \cdot (\log n)^k)$. Finally, finding a smallest classifier over the predicates Φ^* returned by the FindMinCover procedure is very efficient because of the small size of Φ^*. Therefore, the overall complexity of Algorithm 3.3 is close to $O(m^2 \cdot k^2 \cdot n^2 \cdot r \cdot (\log n)^k)$ in practice. Going back to Algorithm 3.1, since the loop in lines $8 - 12$ iterates over a small number of column extractors, the overall empirical complexity of our synthesis algorithm is determined by the complexity of the Algorithm 3.3.

3.6 Implementation

We have implemented our synthesis algorithm in a tool called Mitra, which consists of approximately $7,000$ lines of Java code. As shown in Figure 3.16, Mitra includes a domain-agnostic synthesis core (referred to as Mitra-core) and a set of domain-specific plug-ins. Specifically, Mitra-core accepts input-output examples in the form of (HDT, table) pairs and outputs a program over the DSL shown in Figure 3.7. The goal of a Mitra plug-in is to (1) convert the input document to our internal HDT representation, and (2) translate the program synthesized by Mitra-core to a target DSL. We have currently implemented two domain-specific plug-ins, called Mitra-xml and Mitra-json, for XML and JSON documents respectively. Specifically, Mitra-xml outputs programs written in XSLT, and Mitra-json generates

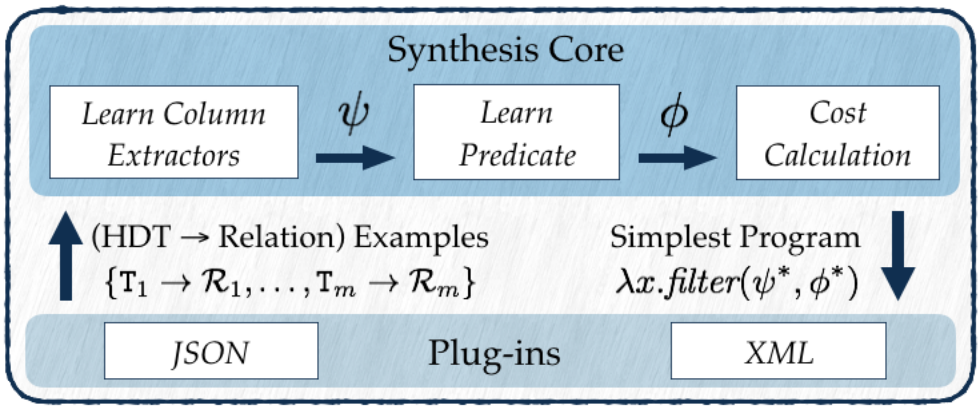


Figure 3.16: Architecture of Mitra

JavaScript programs. Mitra can be easily extended to handle other forms of hierarchical documents (e.g., HTML and HDF) by implementing suitable plug-ins.

3.6.1 Cost function

Recall from Section 3.5 that our synthesis algorithm uses a heuristic cost function θ to choose the simplest program among all possible solutions that are consistent with the provided input-output examples. The cost function that we use in our implementation ranks programs based on the complexity of their predicates and column extractors and returns a program with the lowest cost. Given two programs P_1, P_2, our cost function assigns P_1 (resp. P_2) a lower cost if it uses fewer atomic predicates than P_2 (resp. P_1). If P_1 and P_2 use the same number of atomic predicates, then the cost function assigns a lower cost to the program that uses fewer constructs in the column extractors.

3.6.2 Program Optimization

While our DSL is designed to facilitate synthesis, programs in this language can be inefficient: In particular, the synthesized programs generate a (potentially large) intermediate table and then filter out the undesired tuples. To avoid inefficiencies caused by this design choice, MITRA-CORE optimizes the synthesized programs by avoiding the generation of intermediate tables whenever possible. In particular, consider a synthesized program of the form $\lambda x.\mathit{filter}(\pi_1 \times \pi_2, \phi)$. Instead of first taking the cross-product of π_1, π_2 and then filtering out undesired tuples, we optimize the synthesized program in such a way that the optimized program directly generates the final table by using ϕ to guide table generation. More specifically, we use ϕ to find a prefix subprogram π^* that is shared by π_1, π_2 with the property that any subsequent execution of the remaining parts of π_1, π_2 from any node in $[\![\pi^*]\!]$ yields tuples that are guaranteed to satisfy ϕ. Therefore, the optimized program avoids a post-filtering step and directly generates the output table. A more detailed explanation of this optimization can be found in Appendix B.

3.6.3 Handling Full-fledged Databases

The synthesis algorithm that we described in Section 3.5 generates programs for converting a single HDT to a relational table. However, in practice, we would like to use MITRA to convert XML and JSON datasets to a complete database with a given schema. This transformation can be performed by invoking MITRA multiple times (once for each target table) and annotating

primary and foreign keys of database tables.

MITRA ensures that the synthesized program obeys primary and foreign key constraints by performing a post-processing step. [6] To ensure that the primary key uniquely identifies a given row in the table, the synthesized program generates primary keys as follows: If a given row in the database table is constructed from nodes $n_1, \ldots, n_k$ in the input tree, then we generate its primary key using an injective function $f(n_1, \ldots, n_k)$. Since each row in the table is constructed from a unique list of tree nodes, the generated primary key is guaranteed to be unique for each row as long as f is injective. In our implementation, f simply concatenates the unique identifiers for each tree node.

In order to ensure that a foreign key in table T refers to a primary key in the table T', the synthesized program for table T must use the same function f to generate the foreign keys. In particular, to generate the foreign key for a given row r constructed from list of tree nodes $n_1, \ldots, n_k$, we need to find the tree nodes $n'_1, \ldots, n'_m$ that are used to construct the corresponding row r' in T'. For this purpose, we learn m different (node extractor, node) pairs (χ_j, n_{t_j}) such that $\chi_j(n_{t_j})$ yields n'_j for all rows in the output examples for T and T'. Finally, for a given row r in T, we then generate the foreign key for r as $f(\chi_1(n_{t_1}), \ldots, \chi_m(n_{t_m}))$. This strategy ensures that the foreign and primary

[6] If the primary and foreign keys come from the input data set, we assume that the dataset already obeys these constraints. Hence, the following discussion assumes that the primary and foreign keys do not appear in the input dataset.

key constraints are satisfied as long as the learnt node extractors are correct.

3.7 Evaluation

To evaluate Mitra, we perform experiments that are designed to answer the following questions:

Q1. How effective is Mitra at synthesizing tree-to-table transformation programs?

Q2. Can Mitra be used to migrate real-world XML and JSON datasets to the desired relational database?

Q3. Are the programs synthesized by Mitra fast enough to automate real-world data transformation tasks?

To answer these questions, we perform two sets of experiments: The first experiment evaluates Mitra on tree-to-table transformation tasks collected from StackOverflow, whereas the second experiment evaluates Mitra on real-world datasets. Both experiments are conducted on a MacBook Pro with 2.6 GHz Intel Core i5 processor and 8 GB of 1600 MHz DDR3 memory running OS X version 10.12.5.

3.7.1 Accuracy and Running Time

3.7.1.1 Setup

To perform our first experiment, we collected 98 tree-to-table transformation tasks from StackOverflow by searching for relevant keywords (e.g.,

"JSON to database", "XML shredding", etc.). Among these 98 benchmarks, 51 involve XML documents, while 47 involve JSON files.

Since Mitra requires input-output examples, we obtained the input XML/JSON file directly from the StackOverflow post. For output examples, we used the table provided in the StackOverflow post if one was present; otherwise, we constructed the desired output table based on the English description included in the post.

For each benchmark, we used Mitra to synthesize a program that performs the given task. We manually inspected the tool's output to check whether the synthesized program performs the desired functionality. Even though any program synthesized by Mitra is guaranteed to satisfy the provided input-output examples, it may not necessarily be the program that the user intended. Whenever the program synthesized by Mitra did not conform to the English description provided in the StackOverflow post, we updated the input-output examples to make them more representative. In our evaluation, we needed to update the original input-output example at most once, and the original examples were sufficient to learn the correct program for the majority of the benchmarks.

3.7.1.2 Results

Table 4.3 summarizes the results of evaluating Mitra on these 98 benchmarks. The first part of the table provides information about our benchmarks, which we categorize into four classes depending on the number of columns

	Benchmarks			Synthesis Time (s)		Input-output Examples				Synthesis Program (Avg.)	
						#Elements		#Rows			
	#Cols	Total	#Solved	Median	Avg.	Median	Avg.	Median	Avg.	#Preds	LOC
XML	≤ 2	17	15	0.34	0.38	12.0	15.9	3.0	4.3	1.0	13.2
	3	12	12	0.63	3.67	19.5	47.7	4.0	3.8	2.0	17.2
	4	12	11	1.25	3.56	16.0	20.5	2.0	2.7	3.1	19.5
	≥ 5	10	10	3.48	6.80	24.0	27.2	2.5	2.6	4.1	23.3
	Total	**51**	**48**	**0.82**	**3.27**	**16.5**	**27.2**	**3.0**	**3.5**	**2.4**	**17.8**
JSON	≤ 2	11	11	0.12	0.27	6.0	7.4	2.0	2.7	0.9	21.3
	3	11	11	0.48	1.13	7.0	10.5	3.0	3.5	2.0	23.0
	4	11	11	0.26	12.10	6.0	7.9	2.0	2.8	3.0	26.5
	≥ 5	14	11	3.20	3.85	6.0	8.1	2.0	2.5	4.9	28.0
	Total	**47**	**44**	**0.31**	**4.33**	**6.0**	**8.5**	**2.0**	**2.9**	**2.7**	**24.7**
Overall		**98**	**92**	**0.52**	**3.78**	**11.0**	**18.7**	**3.0**	**3.2**	**2.6**	**21.6**

Table 3.1: Summary of Mitra's experimental evaluation

in the target table. Specifically, "#Cols" shows the number of columns in each category, and "Total" shows the number of benchmarks in each category. The column labeled "#Solved" shows the number of benchmarks that Mitra was able to solve correctly. Overall, Mitra was able to synthesize the target program for 93.9% of these benchmarks.

The columns under "Synthesis Time" show the median and average

time that Mitra takes to synthesize the desired program in seconds. On average, Mitra synthesizes the target transformation in 3.8 seconds, and the median time is even lower (0.5 seconds). We believe these results demonstrate that Mitra is quite practical.

Next, the section labeled "Input-output Examples" in Table 4.3 describes properties of the provided input-output examples. The two columns under "#Elements" represent the median and average number of elements in the input document, and the"#Rows" columns show the median and average number rows in the output table. Here, "#Elements" corresponds to the number of JSON objects and XML elements. As we can see from Table 4.3, the median number of elements in the input document is 11, and the median number of rows in the input table is 3. These results suggest that Mitra can synthesize the desired program from relatively small input-output examples.

The final section of Table 4.3 describes properties of the synthesized programs. For instance, according to the"Preds" column, the average number of atomic predicates used in predicates ϕ from our DSL is 2.6. More interestingly, the column labeled "LOC" gives the number of lines of code in the synthesized XSLT and Javascript programs. On average, the size of the programs synthesized by Mitra is 21.6 (without including built-in functions, such as the implementation of *getDescendants* or code for parsing the input file).

Table 3.2 compares the number (resp. percentage)of benchmarks for which Mitra was able to synthesize the desired program from the original

	Original Examples		Updated Examples	
	#	%	#	%
XML	29	56.9	22	43.1
JSON	27	57.4	20	42.6
Overall	56	57.1	42	42.9

Table 3.2: Number of original vs updated input-output examples.

input-output examples provided in the StackOverflow post (shown in the section "Original Examples") with the number (resp. percentage) of benchmarks that required updating the input-output examples (shown in the section "Updated Examples"). This table shows that the original examples were sufficient to learn the correct program for 57% of our benchmarks.

3.7.1.3 Limitations

To understand Mitra's limitations, we investigated the 6 benchmarks for which Mitra was not able to synthesize the desired program. We found that the desired program was not expressible in our DSL for 5 of these 6 benchmarks. For example, some of these 5 benchmarks require the use of a conditional in the column extractor, which our DSL does not currently support. For the other benchmark, there exists a DSL program that can perform the desired functionality, but Mitra ran out of memory due to the large number of columns in the target table.

3.7.1.4 Performance

We also evaluated the performance of the synthesized programs by running them on XML documents of size 512 ± 20 MB. In particular, we used the Oxygen XML editor [7] to generate XML files with a given schema and a specified number of elements. Among the 48 XSLT programs generated using Mitra, 46 of them were able to perform the desired transformation within approximately one minute. The running times of these 46 programs range between 8.6 and 65.5 seconds, with a median (resp. average) running time of 20.0 (resp. 23.5) seconds. The other two programs were not able to complete the transformation task within one hour due to inefficiencies in the generated code.

3.7.2 Migration to Relational Database

3.7.2.1 Setup

In our next experiment, we use Mitra to convert real-world XML and JSON datasets to a complete relational database. Our benchmarks for this experiment include the following four well-known datasets:

- **DBLP**, an XML document containing 2 GB of data about published computer science papers [3].
- **IMDB**, a set of JSON documents containing 6.2 GB of data about movies, actors, studios etc [4]. [7]

[7] The raw data from [4] is provided in tab-separated-values (TSV) format, so we converted it to JSON format using an existing program [5].

- **MONDIAL**, an XML document containing 3.6 MB of geographical data [6].
- **YELP**, a set of JSON documents containing 4.6 GB of data about businesses and reviews [11].

We obtained the target relational database schema for each of these datasets from [2], and certified that all of them are normalized. To use Mitra to perform the desired data migration task, we manually constructed small input-output examples. For each table, we provided a single pair of input-output examples, in addition to a list of all primary and foreign keys in the database schema. In particular, the average number of elements in the input examples is 16.6 and the average number of rows in each database table is 2.8. Given a suitable input-output example for each target database table, we then ran Mitra with a time limit of 120 seconds. We then manually inspected the synthesized program and verified its correctness. In this experiment, there was no user interaction; the original examples we provided were sufficient for Mitra to synthesize the desired program. Furthermore, for each target table, we were often able to re-use the input examples (e.g., for IMDB, we used 2 different input examples for the 9 target tables).

3.7.2.2 Results

The results of this experiment are summarized in Table 3.3. The columns under "DB" show the number of tables and total number of attributes in the

Name	Dataset		DB		Synthesis Time(s)		Execution		
	Format	Size	#Tables	#Cols	Tot.	Avg.	#Rows	Tot. Time(s)	Avg. Time(s)
DBLP	XML	1.97 GB	9	39	7.41	0.82	8.312 M	1166.44	129.60
IMDB	JSON	6.22 GB	9	35	33.53	3.72	51.019 M	1332.93	148.10
MONDIAL	XML	3.64 MB	25	120	62.19	2.48	27.158 K	71.84	2.87
YELP	JSON	4.63 GB	7	34	14.39	2.05	10.455 M	220.28	31.46

Table 3.3: Migrating datasets to databases. Summary of using MITRA for migrating real-world datasets to a full DB. The columns labeled "Tot. Time" include time for all database tables, whereas "Avg. Time" is the average time per table.

target database. In particular, the number of database tables range between 7 and 25 and the number of columns range from 34 to 120. [8] Next, the two columns under "Synthesis" show total and average synthesis time in seconds, respectively. Specifically, the average time denotes synthesis time per table, whereas total time aggregates over all database tables. As shown in Table 3.3, average synthesis time ranges between 0.8 to 3.7 seconds per database table.

The last part of Table 3.3 provides information about the execution time of the synthesized program and size of the generated database. Specifically,

[8]The Mondial database consists of a large number of tables and columns because it includes a variety of different geographical and cultural data.

according to the "#Rows" column, the total number of rows in the generated database ranges between $27,000$ and 51 million. The next two rows provide statistics about the execution time of the synthesized programs on the original full dataset. For example, the average time to generate a target database table for datasets in the range $2-6$ GB is between 31 and 148 seconds. These statistics show that the programs generated by Mitra can be used to migrate real-world large datasets.

3.8 Summary

In this chapter, we proposed a new PBE-based method for transforming hierarchically structured data, such as XML and JSON documents, to a relational database. Given a small input-output example, our tool, Mitra, can synthesize XSLT and Javascript programs that perform the desired transformation. The key idea underlying our method is to decompose the synthesis task into two phases for learning the column and row extraction logic separately. Our synthesis algorithm generates column extractors by constructing a certain type of DFA and identifying the lowest-cost word accepted by this DFA. In contrast, our method learns the predicates used in the row extraction logic by reducing the problem to a combination of integer linear programming and logic minimization. We have evaluated our method on examples collected from Stackoverflow as well as real-world XML and JSON datasets. Our evaluation shows that Mitra is able to synthesize the desired programs for 94% of the Stackoverflow benchmarks and for all of the real-world datasets.

Chapter 4

Sqlizer [1]

One of the main advantages of using relational databases is the ease of data retrieval. While extracting data from relational databases is more efficient and less complex than retrieving it from hierarchical datasets such (e.g. XML or JSON documents), it still requires users to write queries in a specific language such as SQL. This means end-users who wish to access data in an underlying database should be knowledgeable about database systems and query languages.

This chapter presents a new technique for automatically synthesizing SQL queries from natural language (NL). At the core of our technique is a new NL-based program synthesis methodology that combines semantic parsing techniques from the NLP community with type-directed program synthesis and automated program repair. Starting with a program sketch obtained using standard parsing techniques, our approach involves an iterative refinement loop that alternates between quantitative type inhabitation and automated sketch repair. We use the proposed idea to build an end-to-end system called Sqlizer that can synthesize SQL queries from natural language. Our method is fully

[1] Parts of this chapter have appeared in [186].

automated, works for any database without requiring additional customization, and does not require users to know the underlying database schema. We evaluate our approach on over 450 natural language queries concerning three different databases, namely MAS, IMDB, and YELP. Our experiments show that the desired query is ranked within the top 5 candidates in close to 90% of the cases.

The rest of this chapter is organized as follows: We first introduce our approach in Section 4.1 and provide an overview of it through a simple motivating example (Section 4.2). Then, we present our general methodology for synthesizing programs from natural language descriptions (Section 4.3). After providing some background on a variant of relational algebra (Section 4.4), we then show how to instantiate each of the components of our general synthesis methodology in the context of SQL synthesis (Sections 4.5, 4.6, 4.7). We discuss our implementation in Section 4.8 and present our empirical evaluation in Section 4.9. Finally, we discuss the limitations of our system in Section 4.10 and summarize this chapter in Section 4.11.

4.1 Introduction

A promising application domain for program synthesis from informal specifications is the automated synthesis of database queries. Although many end-users need to query data stored in some relational database, they typically lack the expertise to write complex queries in declarative query languages such as SQL. As a result, there has been a flurry of interest in automatically

synthesizing SQL queries from informal specifications [194, 176, 59, 113, 144].

Existing techniques for automatically synthesizing SQL queries fall into two different classes, namely example-based approaches and those based on natural language. Programming-by-example techniques, such as Scythe [176] and SQLSynthesizer [194], require the user to present a miniature version of the database together with the expected output. A shortcoming of such example-directed techniques is that they require the user to be familiar with the database schema. Furthermore, because realistic databases typically involve many tables, it can be quite cumbersome for the user to express their intent using input-output examples.

On the other end of the spectrum, techniques that can generate SQL queries from natural language (NL) descriptions are easier for end-users but more difficult for the underlying synthesizer, as natural language is inherently ambiguous. Existing NL-based techniques try to achieve high precision either by training the system on a specific database [192, 170] or requiring interactive guidance from the user [113]. Hence, existing NL-based techniques for synthesizing SQL queries are either not fully automatic or not database-agnostic (i.e., require customization for each database).

In this chapter, we present a novel technique, and its implementation in a tool called Sqlizer, for synthesizing SQL queries from English descriptions. By marrying ideas from the NLP community with type-directed synthesis and repair techniques from the programming languages community, our proposed approach overcomes many of the disadvantages of existing techniques. Specifically, our

method is fully automatic (i.e., it does not require guidance from the user) and database-agnostic (i.e., it does not require database-specific training or customization). As we show experimentally, SQLIZER achieves high precision across multiple different databases and significantly outperforms NALIR [113], a state-of-the-art system that won a best paper award at VLDB'14.

From a technical perspective, our approach achieves these results by combining three novel and synergistic ideas in a confidence-driven refinement loop:

Sketch generation semantic parsing. As a first step, our method uses standard semantic parsing techniques [97, 116, 96] from the NLP community to translate the user's English description into a so-called *query sketch (skeleton)*. Since a query sketch only specifies the shape – rather than the full content – of the query (e.g., join followed by selection followed by projection), the semantic parser does not need to be trained on the target database. Hence, by using the semantic parser to generate a query sketch rather than a full-fledged SQL query, we can translate the user's English description into a suitable formal representation *without requiring any database-specific training*. This property of being database-agnostic is very useful because our system does not need additional training data for each new database that a user wishes to query.

Type-directed sketch completion. Given a query sketch containing holes, our technique employs type-directed program synthesis to complete the sketch into a well-typed SQL query. However, because there are typically many

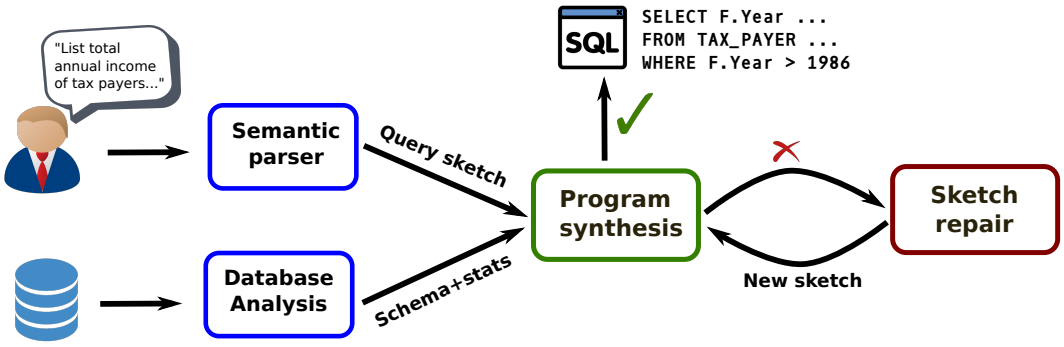


Figure 4.1: Schematic overview of SQLIZER's approach

well-typed completions of the sketch, our approach assigns a *confidence score* to each possible completion using both the schema and the contents of the database.

Sketch refinement using repair. Since users are typically not familiar with the underlying data organization, the initial query sketches generated using semantic parsing may not accurately reflect the structure of the target query. Hence, there may not be any well-typed, high-confidence completions of the sketch. For example, consider a scenario in which the user believes that the desired data is stored in a single database table, although it in fact requires joining two different tables. Since the user's English description does not adequately capture the structure of the desired query, the initial query sketch needs to be repaired. Our approach deals with this challenge by (a) performing fault localization to pinpoint the likely cause of the error, and (b) using a database of "repair tactics" to refine the initial sketch.

Figure 4.1 gives a schematic overview illustrating the interaction between the three key ideas outlined above. Given the user's natural language description, SQLIZER first generates the top k most likely query sketches Q using semantic parsing, and, for each query skeleton $q \in Q$, it tries to synthesize a well-typed, high-confidence completion of q. If no such completions can be found, SQLIZER tries to identify the root cause of failure and automatically repairs the suspect parts of the sketch. Once SQLIZER enumerates all high-confidence queries that can be obtained using at most n repairs on q, it then moves on to the next most-likely query sketch. At the end of this *parse-synthesize-repair* loop, SQLIZER ranks all queries based on their confidence scores and presents the top m results to the user.

We have evaluated our approach on 455 queries involving three different databases, namely MAS (Microsoft Academic Search), IMDB, and YELP. Our evaluation shows that the desired query is ranked within SQLIZER's top 5 solutions approximately 90% of the time and within top 1 close to 80% of the time. We also compare SQLIZER against a state-of-the-art NLIDB tool, NALIR, and show that SQLIZER performs significantly better across all three databases, including on the data set that is used for evaluating NALIR.

4.1.1 The General Idea

While our target application domain in this chapter is relational databases, we believe that our proposed ideas could be applicable in other domains that require synthesizing programs from natural language descriptions.

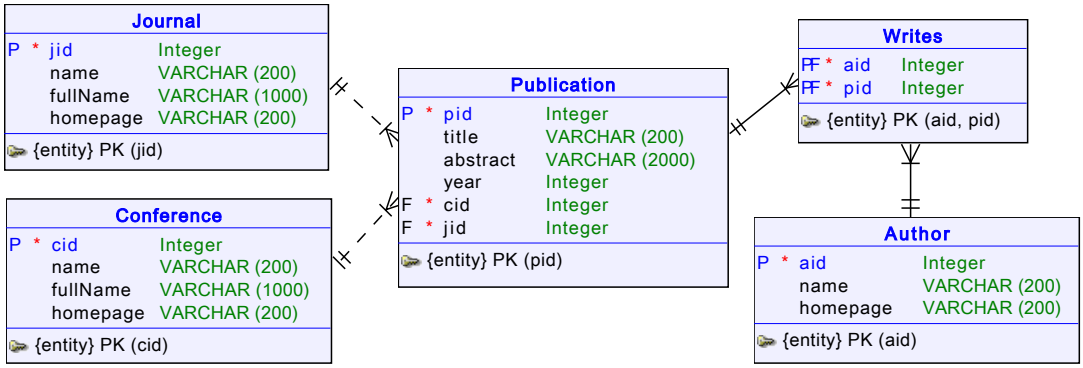


Figure 4.2: Simplified schema for MAS database. MAS stands for Microsoft Academic Search.

Specifically, given a program sketch generated using standard NLP techniques, we propose a confidence-driven synthesis methodology that uses a form of *quantitative type inhabitation* to assign a confidence score to each well-typed sketch completion. While the specific technique used for assigning confidence scores is inherently domain-specific, the idea of assigning confidence scores to type inhabitants is not. Given a domain-specific method for performing quantitative type inhabitation and a database of domain-specific repair techniques, the *parse-synthesize-repair* loop proposed in this chapter can be instantiated in many other settings where the goal is to synthesize programs from natural language.

4.2 Overview

In this section, we give a high-level overview of our technique with the aid of a simple motivating example. Figure 4.2 shows the relevant portion of the

schema for the Microsoft Academic Search (MAS) database, and suppose that we would like to synthesize a database query to retrieve the number of papers in OOPSLA 2010. To use our tool, the user provides an English description, such as *"Find the number of papers in OOPSLA 2010"*. We now outline the steps taken by SQLIZER in synthesizing the desired SQL query.

Sketch generation. Our approach first uses a semantic parser to generate the top k most-likely program sketches. For this example, the highest-ranked query sketch returned by the semantic parser is the following [2]:

```
SELECT count(?[papers]) FROM ??[papers]
WHERE ? = "OOPSLA 2010"
```

Here, `??` represents an unknown table, and `?`'s represent unknown columns. Where present, the words written in square brackets represent so-called "hints" for the corresponding hole. For example, the second hint in this sketch indicates that the table represented by `??` is semantically similar to the English word "papers".

First iteration. Starting from the above sketch $\mathcal{S}$, SQLIZER enumerates all well-typed completions q_i of $\mathcal{S}$, together with a score for each q_i. In this case, there are many possible well-typed completions of $\mathcal{S}$, however, none of the q_i's meet our confidence threshold. For instance, one of the reasons why SQLIZER

[2] We actually represent query sketches using an extended version of relational algebra. In this section, we present query sketches using SQL for easier readability.

fails to find a high-confidence query is that there is no entry called "OOPSLA 2010" in any of the database tables.

Next, SQLIZER performs fault localization on $\mathcal{S}$ to identify the root cause of failure (i.e., not meeting confidence threshold). In this case, we determine that the likely root cause is the predicate `? = "OOPSLA 2010"` since there is no database entry matching "OOPSLA 2010", and our synthesis algorithm has assigned a low confidence score to this term. Next, we repair the sketch by splitting the where clause into two separate conjuncts. As a result, we obtain the following refinement $\mathcal{S}'$ of the initial sketch $\mathcal{S}$:

```
SELECT count(?[papers]) FROM ??[papers]
WHERE ? = "OOPSLA" AND ? = 2010
```

Second iteration. Next, SQLIZER tries to complete the refined sketch $\mathcal{S}'$ but it again fails to find a high-confidence completion of $\mathcal{S}'$. In this case, the problem is that there is no single database table that contains both the entry "OOPSLA" as well as the entry "2010". Going back to fault localization, we now determine that the most likely problem is the term `??[papers]`, and we try to repair it by introducing a join. As a result, the new sketch $\mathcal{S}''$ now becomes:

```
SELECT count(?[papers]) FROM  ??[papers] JOIN  ??
ON ? = ?  WHERE ? = "OOPSLA" AND ? = 2010
```

Third iteration. After going back to the sketch completion phase a third time, we are now able to find a high-confidence instantiation q of $\mathcal{S}''$. In this case, the highest ranked completion of $\mathcal{S}''$ corresponds to the following query:

```
SELECT count(Publication.pid)
FROM Publication JOIN Conference
     ON Publication.cid = Conference.cid
WHERE Conference.name = "OOPSLA"
      AND Publication.year = 2010
```

This query is indeed the correct one, and running it on the MAS database yields the number of papers in OOPSLA 2010.

4.3 General Synthesis Methodology

Before describing our specific technique for synthesizing SQL code from English queries, we first explain our general methodology for synthesizing programs from natural language descriptions. As mentioned in Section 3.1, our general synthesis methodology involves three components, namely (a) sketch generation using semantic parsing, (b) sketch completion using quantitative type inhabitation, and (c) sketch refinement using repair.

The high-level structure of our synthesis methodology is described in pseudo-code in Algorithm 4.1. The algorithm takes as input a natural language description Ω of the program to be synthesized, a type environment Γ, and a confidence threshold γ. Since our synthesis algorithm assigns confidence

Algorithm 4.1 General synthesis methodology in SQLIZER

```
1: procedure SYNTHESIZE(Q, Γ, γ)
2:     Input: natural language query Q, type environment Γ, confidence threshold γ
3:     Output: A list of synthesized programs and their corresponding confidence scores
4:     Sketches := SEMANTICPARSE(Q)                          ▷ Sketch generation
5:     Programs := [ ];
6:     for all top k ranked S ∈ Sketches do
7:         loop n times
8:             θ := FINDINHABITANTS (S, Γ)                   ▷ Type-directed sketch completion
9:             needRepair := true
10:            for all (I_i, P_i) ∈ θ do
11:                if P_i > γ then Programs.add(I_i, P_i); needRepair := false;
12:            if ¬needRepair then break
13:            F := FAULTLOCALIZE(S, Γ, ∅)                   ▷ Sketch refinement
14:            if F = null then break
15:            S := S[FINDREPAIR(F)/F]
16:    return Programs
```

scores to generated programs, the cut-off γ is used to identify programs that do not meet some minimum confidence threshold. The output of the synthesis procedure is a list of synthesized programs, together with their corresponding confidence score.

Given an English description of the program, the first step in our synthesis methodology is to generate a ranked list of program sketches using semantic parsing [97, 116, 96]. Semantic parsing is a standard technique from the NLP

community that can be used to convert an English utterance into a logical form, which is defined according to a formal context-free grammar. In this chapter, we use *program sketches* [165, 166] as our logical form representation because they allow us to translate the NL description into a program draft while omitting several low-level details that may not be accurately captured by the user's English description. As standard, a *program sketch* in this context is a partial program with holes [165, 166]. However, because our program sketches are created from English descriptions, we additionally allow each hole in the sketch to contain a natural language *hint* associated with it. The idea is to use these natural language hints for assigning confidence scores during sketch completion.

Given the top k query sketches generated by the semantic parser, the next step is to complete each sketch $\mathcal{S}$ such that it is well-typed with respect to our type environment Γ. For instance, in the context of SQL, the type environment corresponds to the database schema, and the goal of sketch completion is to find a relational algebra term that is well-typed with respect to the underlying database schema. In essence, the use of type information allows our synthesis methodology to perform logical reasoning that complements the probabilistic reasoning performed by the semantic parser. However, because there are typically many well-typed completions of a given sketch, we would like to predict which term is the *most likely* completion. We refer to this problem as *quantitative type inhabitation*: Given a type τ, a type environment Γ, and some "soft constraints" $\mathcal{C}$ on the term to be synthesized, what is our confidence

$\mathcal{P}_i$ that $\mathcal{I}_i$ is the inhabitant of τ with respect to hard constraints Γ and soft constraints $\mathcal{C}$? These soft constraints include natural language hints embedded inside the sketch as well as any other domain-specific knowledge. For example, in the context of database queries, we also utilize the contents of the database when assigning confidence scores to relational algebra terms.

Now, if we fail to find any well-typed completions of the sketch that meet our confidence threshold γ, our algorithm goes into a refinement loop that alternates between repair and synthesis (the inner loop at lines 7–15 in Algorithm 4.1). Given a program sketch $\mathcal{S}$ for which we cannot find a high-confidence completion, the fault localization procedure (line 13) returns a *minimal subterm* of the sketch that does not meet our confidence threshold γ. Given a faulty subterm $\mathcal{F}$ of sketch $\mathcal{S}$, we then consult a database of *domain-specific repair tactics* to find a new subterm $\mathcal{F}'$ that can be used to replace $\mathcal{F}$. For instance, in the domain of SQL query synthesis, the repair tactics introduce join operators, aggregate functions, or additional conjuncts in predicates depending on the shape of the faulty sub-term. If there are multiple repair tactics that apply, we can either arbitrarily choose one or try each of them in turn. Since this refinement process may, in general, continue ad infinitum, our algorithm allows the user to specify a value n that controls the number of refinement steps that are allowed.

$$
\begin{aligned}
T &:= \Pi_L(T) \mid \sigma_\phi(T) \mid T\,{}_c\!\bowtie_c T \mid t \\
L &:= L, L \mid c \mid f(c) \mid g(f(c), c) \\
E &:= T \mid c \mid v \\
\phi &:= \phi\ lop\ \phi \mid \neg\phi \mid c\ op\ E \\
op &:= \leq \mid < \mid = \mid > \mid \geq \\
lop &:= \wedge \mid \vee
\end{aligned}
$$

Figure 4.3: Grammar of Extended Relational Algebra. Here, t, c denote table and column names; f denotes an aggregate function, and v denotes a value.

4.4 Extended Relational Algebra

In the rest of this chapter, we will show how to apply the synthesis methodology outlined in Section 4.3 to the problem of synthesizing database queries from natural language. However, since our approach synthesizes database queries in a variant of relational algebra, we first describe our target language. Note that it is straightforward to translate our extended relational algebra to declarative query languages, such as SQL.

Our target language for expressing database queries is presented in Figure 4.3. Here, *relations* are denoted as T and include *tables* t stored in the database or *views* obtained by applying the relational algebra operators, projection (Π), selection (σ), and join ($\bowtie$). As standard, projection $\Pi_L(T)$ takes a relation T and a column list L and returns a new relation that only contains the columns in L. The selection operation $\sigma_\phi(T)$ yields a new relation that only contains rows satisfying ϕ in T. The join operation $T_1\,{}_{c_1}\!\bowtie_{c_2} T_2$

composes two relations T_1, T_2 such that the result contains exactly those rows of $T_1 \times T_2$ satisfying $c_1 = c_2$, where c_1, c_2 are columns in T_1, T_2 respectively. Please observe that the grammar from Figure 4.3 allows nested queries. For instance, selections can occur within other selections and joins as well as inside predicates ϕ.

In the rest of this chapter, we make a few assumptions that simplify our technical presentation. First, we assume that every column in the database has a unique name. Note that we can easily enforce this restriction by appending the table name to each column name. Second, we only consider *equi-joins* because they are the most commonly used join operator; however, our techniques can also be extended to other kinds of join operators (e.g., θ-join).

Unlike standard relational algebra, the relational algebra variant shown in Figure 4.3 also allows aggregate functions as well as a group-by operator. For conciseness, aggregate functions $f \in \mathit{AggrFunc} = \{\mathit{max}, \mathit{min}, \mathit{avg}, \mathit{sum}, \mathit{count}\}$ are specified as a subscript in the projection operation. In particular, $\Pi_{f(c)}(T)$ yields a single aggregate value obtained by applying f to column c of relation T. Similarly, group-by operations are also specified as a subscript in the projection operator. Specifically, $\Pi_{g(f(c_1),c_2)}(T)$ divides rows of T into groups g_i based on values stored in column c_2 and, for each g_i, it yields the aggregate value $f(c_1)$.

Example 4.1. *Consider the "Grades" and "Courses" tables from Figure 4.4, where column names with suffix "_fk" indicate foreign keys. Here,* $\Pi_{\text{avg(score)}}(\text{Grades})$ *evaluates to 85, and*

id	name	score	cid_fk
1	John	60	101
2	Jack	80	102
3	Jane	80	103
4	Mike	90	104
5	Peter	100	103
6	Alice	100	104

(a) Grades

cid	cname	dept
101	C1	CS
102	C2	EE
103	C3	CS
104	C4	EE

(b) Courses

Figure 4.4: Example 4.1 tables

$\Pi_{\text{g(avg(score), dept)}}(\text{Grades}_{cid\_fk} \bowtie_{cid} \text{Courses})$ *yields the following table:*

dept	*avg(score)*
CS	*80*
EE	*90*

To provide an example of nested queries, suppose that a user wants to retrieve all students with the highest score. We can express this query as $\Pi_{\text{name}}(\sigma_{\text{score}=\Pi_{\text{max(score)}}(\text{Grades})}(\text{Grades}))$*. For the tables from Figure 4.4, this query yields a table with two rows, Peter and Alice.*

4.5 Sketch Generation Using Semantic Parsing

Recall from Section 4.3 that the first step of our synthesis methodology is to generate a program sketch using semantic parsing. In this section, we provide some background on semantic parsing and describe our parser for generating a query sketch from the user's English description.

4.5.1 Background on Semantic Parsing

The goal of a semantic parser is to map a sentence in natural language to a so-called *logical form* which represents its meaning. A logical form is an unambiguous artificial language specified using a context-free grammar. Previous work on semantic parsing has used various logical form representations, including lambda calculi [39], database query languages [192], and natural logics [121].

Similar to traditional parsers used in compilers, a semantic parser is specified by a context-free grammar and contains a set of reduction rules describing how to derive non-terminal symbols from token sequences. Given an English sentence S, the parse is successful if the designated root non-terminal can be derived from S.

Since semantic parsers deal with natural language, they must overcome two challenges that do not arise in parsers for formal languages. First, multiple English phrases (e.g., "forty two") can correspond to a single token (e.g., 42); so a semantic parser must be capable of recognizing phrases and mapping them accurately into tokens. To address this challenge, semantic parsers typically include a linguistic processing module to analyze the sentence and detect such phrases based on part-of-speech tagging and named entity recognition. Second, since natural language is inherently ambiguous, one can often derive multiple logical forms from an English sentence. Modern semantic parsers address this challenge by using statistical methods to predict the most likely derivation for a given utterance. Given a set of training data consisting of pairs of English

$$
\begin{aligned}
\chi &:= \Pi_\kappa(\chi) \mid \sigma_\psi(\chi) \mid \chi_{?h} \bowtie_{?h} \chi \mid ??h \\
\kappa &:= \kappa, \kappa \mid ?h \mid f(?h) \mid g(f(?h), ?h) \\
\eta &:= \chi \mid ?h \mid v \\
\psi &:= \psi \; lop \; \psi \mid \neg\psi \mid ?h \; op \; \eta \\
op &:= \leq \mid < \mid = \mid > \mid \geq \\
lop &:= \wedge \mid \vee
\end{aligned}
$$

Figure 4.5: Sketch grammar. Here, v denotes a value, h represents a natural language hint, and f is an aggregate function.

sentences and their corresponding logical form, the idea is to train a statistical model to predict the likelihood that a given English sentence is mapped to a particular logical form. Hence, the output of a semantic parser is a list of logical forms x_i, each associated with a probability that the English sentence corresponds to x_i.

4.5.2 SQLIZER's Semantic Parser

The logical forms used in our synthesis methodology take the form of program sketches containing unknown expressions with natural language hints. In essence, the use of program sketches as our logical form representation allows the semantic parser to generate a high-quality intermediate representation even when we have modest amounts of training data available. In the context of generating database queries from natural language, the use of query sketches allows our technique to work well *without requiring any database-specific training.*

$$
\begin{array}{rcl}
\beta & := & \mathit{Number}\,\mathbb{N} \mid \mathit{Bool}\,\mathbb{B} \mid \mathit{String}\,\mathbb{S} \mid \cdots \\
\tau & := & \beta \mid \{(c_1 : \beta_1), \cdots, (c_n : \beta_n)\} \\
\Gamma & :: & \mathit{Table} \to \tau \\
p & :: & \{\nu : \mathit{double} \mid 0 \leq \nu \leq 1\} \\
\mathit{sim} & :: & \mathbb{S} \times \mathbb{S} \to p \\
P_{\bowtie} & :: & \mathit{Column} \times \mathit{Column} \to p \\
P_{\phi} & :: & \mathit{Column} \times \mathit{Value} \to p
\end{array}
$$

Figure 4.6: Symbols used in sketch completion

Figure 4.5 defines *query sketches* that are used as our logical form representation in the database query synthesis domain. At a high level, a query sketch χ is a relational algebra term with missing table and column names. In particular, $?h$ represents an unknown column with *hint* h, which is just a natural language description of the unknown. Similarly, $??h$ represents an unknown table name with corresponding hint h. If there is no hint associated with a hole, we simply write ? for columns and ?? for tables.

To map English sentences to query sketches, we have implemented our own semantic parser on top of the SEMPRE framework [29], which is a toolkit for building semantic parsers. For the linguistic processor, we leverage the pre-trained models of the Stanford CoreNLP [124] library for part-of-speech tagging and named entity recognition.

Given an utterance u, our parser generates all possible query sketches $\mathcal{S}_i$ and assigns each $\mathcal{S}_i$ a score that indicates the likelihood that $\mathcal{S}_i$ is the intended

interpretation of u. This score is calculated based on a set of pre-defined features. More precisely, given an utterance u and weight vector w, the parser maps each query sketch $\mathcal{S}_i$ to a d-dimensional feature vector $\phi(u, \mathcal{S}_i) \in \mathbb{R}^d$ and computes the likelihood score for $\mathcal{S}_i$ as the weighted sum of its features:

$$score(u, \mathcal{S}_i) \;=\; \vec{w} \;.\; \phi(u, \mathcal{S}_i) \;=\; \sum_{j=1}^{d} w_j \cdot \phi(u, \mathcal{S}_i)_j$$

SQLIZER uses approximately 40 features that it inherits from the SEMPRE framework. Examples of features include the number of grammar rules used in the derivation, the length of the matched input, whether a particular rule was used in the derivation, the number of skipped words in a part-of-speech tag etc.

4.6 Type-Directed Sketch Completion

Given a program sketch $\mathcal{S}$ and a type environment (in our case, database schema) Γ, sketch completion refers to the problem of finding an expression e such that e is well-typed with respect to Γ. In essence, this is a type inhabitation problem in that we can view each program sketch as defining a type. However, rather than finding *any* inhabitant of $\mathcal{S}$, we would like to find an inhabitant e of $\mathcal{S}$ that has a high probability of being the program that the user has in mind. One of the key ideas in our approach is to use natural language hints embedded in sketch $\mathcal{S}$ as well as any domain-specific background knowledge to associate a confidence score $\mathcal{P}_i$ with every inhabitant $\mathcal{I}_i$ of a type τ with respect to the type environment Γ. As mentioned earlier, we refer to this problem as

quantitative type inhabitation.

In the specific case of the database query synthesis problem, we make use of the following high-level insights to determine the confidence score of each type inhabitant:

- *Names of schema elements:* Since our query sketches contain natural language hints for each hole, we can utilize table and column names in the database schema to assign confidence scores.

- *Foreign and primary keys:* Since foreign keys provide links between data in two different database tables, join operations that involve foreign keys have a higher chance of being the intended term.

- *Database contents:* Our approach also uses the contents of the database when assigning scores to queries. For instance, a candidate term $\sigma_\phi(T)$ is relatively unlikely to occur in the target query if there are no entries in relation T satisfying predicate ϕ.[3]

Using these domain-specific heuristics in the query synthesis context, Figures 4.7 and 4.8 describe our quantitative type inhabitation rules. Specifically, the top-level sketch completion rules derive judgments of the form $\Gamma \vdash \chi \Downarrow T : \tau, p$ where Γ is a type environment mapping each table t in the

[3]This assumption may not hold in all contexts. However, since SQLIZER is intended as a question answering system, we believe this assumption makes sense under potential deployment scenarios of a tool like SQLIZER.

database to its corresponding type. The meaning of this judgment is that sketch χ instantiates to a relational algebra term T of type τ with confidence score $p \in [0, 1]$. The higher the score p, the more likely it is that T is a valid completion of sketch χ.

Figure 4.8 presents the helper rules used for finding inhabitants of so-called *specifiers*. A specifier ω of a sketch χ is any subterm of χ that does not correspond to a relation. For example, the specifier for $\Pi_\kappa(\chi)$ is κ, and the specifier for $\sigma_\psi(\chi)$ is ψ. The instantiation rules for specifiers are shown in Figure 4.8 using judgements of the form:

$$\Gamma, \tau \vdash_s \omega \Downarrow Z : \tau', p$$

Here, type τ used on the left-hand side of the judgment denotes the type of the table that ω is supposed to qualify. For instance, if the parent term of ω is $\Pi_\omega(\chi)$, then τ represents the type of relation χ. Hence, the meaning of this judgment is that, under the assumption that ω qualifies a relation of type τ, then ω instantiates to a term Z of type τ' with score p.

4.6.1 Inhabitation Rules for Relation Sketches

Let us first consider the quantitative type inhabitation rules from Figure 4.7. Given a sketch $??h$ indicating an unknown database table with hint h, we can, in principle, instantiate $??$ with any table t in the database (i.e., t is in the domain of Γ). However, as shown in the first rule from Figure 4.7, our approach uses the hint h to compute the likelihood that t is the intended

$$\frac{t \in dom(\Gamma) \quad p = sim(h, t)}{\Gamma \vdash ??h \Downarrow t : \Gamma(t),\ p} \qquad \text{(Table)}$$

$$\frac{\Gamma \vdash \chi \Downarrow T : \tau,\ p_1 \quad \Gamma, \tau \vdash_s \kappa \Downarrow L : \tau_1,\ p_2}{\Gamma \vdash \Pi_\kappa(\chi) \Downarrow \Pi_L(T) : \tau_1,\ p_1 \otimes p_2} \qquad \text{(Proj)}$$

$$\frac{\Gamma \vdash \chi \Downarrow T : \tau,\ p_1 \quad \Gamma, \tau \vdash_s \psi \Downarrow \phi : \mathbb{B},\ p_2}{\Gamma \vdash \sigma_\psi(\chi) \Downarrow \sigma_\phi(T) : \tau,\ p_1 \otimes p_2} \qquad \text{(Sel)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \chi_1 \Downarrow T_1 : \tau_1,\ p_1 \quad \Gamma \vdash \chi_2 \Downarrow T_2 : \tau_2,\ p_2 \\ \Gamma, \tau_1 \vdash_s ?h_1 \Downarrow c_1 : \{(c_1, \beta)\},\ p_3 \\ \Gamma, \tau_2 \vdash_s ?h_2 \Downarrow c_2 : \{(c_2, \beta)\},\ p_4 \\ p = p_1 \otimes p_2 \otimes p_3 \otimes p_4 \otimes P_{\bowtie}(c_1, c_2)\end{array}}{\Gamma \vdash \chi_{1\,?h_1} \bowtie_{?h_2} \chi_2 \Downarrow T_{1\,c_1} \bowtie_{c_2} T_2 : \tau_1 \cup \tau_2,\ p} \qquad \text{(Join)}$$

Figure 4.7: Inference rules for relations

completion of ??. Specifically, we use the *sim* procedure to compute a similarity score between hint h and table name t using Word2Vec [129], which uses a two-layer neural net to group similar words together in vector-space.

Next, let us consider the *Proj*, *Sel* rules from Figure 4.7. Given a sketch of the form $\Pi_\kappa(\chi)$ (resp. $\sigma_\psi(\chi)$), we first recursively instantiate the sub-relation χ to T. Now, observe that the columns used in specifiers κ and ψ

can only refer to columns in T; hence, we use the type τ of T when instantiating specifiers κ, ψ. Now, assuming χ instantiates to T with score p_1 and κ (resp. ψ) instantiates to L (resp. ϕ) with score p_2, we need to combine p_1 and p_2 to determine a score for $\Pi_L(T)$ (resp. $\sigma_\phi(T)$). Towards this goal, we define an operator $\otimes$ that is used to compose different scores. While there are many possible ways to compose scores, our implementation defines $\otimes$ as the geometric mean of $p_1, \ldots, p_n$:

$$p_1 \otimes ... \otimes p_n = \sqrt[n]{p_1 \times ... \times p_n}$$

Observe that our definition of $\otimes$ is not the standard way to combine probabilities (i.e., standard multiplication). We have found that this definition of $\otimes$ works better in practice, as it does not penalize long queries and allows a more meaningful comparison between different-length queries. However, one implication of this choice is that the scores of all possible completions do not add up to 1. Hence, our confidence scores are not "probabilities" in the technical sense, although they are in the range $[0, 1]$.

Finally, let us consider the *Join* rule for completing sketches of the form $\chi_1 {}_{?h_1}\bowtie_{?h_2} \chi_2$. As before, we first complete the nested sketches χ_1 and χ_2, and then instantiate $?h_1$ and $?h_2$ under the assumption that χ_1, χ_2 have types τ_1, τ_2 respectively. The function $P_\bowtie(c_1, c_2)$ used in the *Join* rule assigns a higher score to term $T_1 {}_{c_1}\bowtie_{c_2} T_2$ if column c_1 is a foreign key referring to column c_2 in table T_2 (or vice versa). Intuitively, if c_1 in table T_1 is a foreign key referring to c_2 in Table T_2, then there is a high probability that the term $T_1 {}_{c_1}\bowtie_{c_2} T_2$ appears in the target query. More precisely, we define the $P_\bowtie$ function as

follows, where ϵ is a small, non-zero constant:

$$P_{\bowtie}(c_1, c_2) = \begin{cases} 1 - \epsilon & \text{if } c_1 \text{ is a foreign key referring to } c_2 \text{ (or vice versa)} \\ \epsilon & \text{otherwise} \end{cases}$$

4.6.2 Inhabitation rules for specifiers

In our discussion so far, we ignored how to find inhabitants of specifiers and how to assign confidence scores to them. We now consider the rules from Figure 4.8 that address this problem.

In the simplest case, a specifier is a column of the form $?h$. As shown in the *Col* rule of Figure 4.8, we must first ensure that the candidate inhabitant c is actually a column of the table associated with the parent relation (i.e., $(c, \beta) \in \tau$). In addition to this "hard constraint", we also want to assign a higher confidence score to inhabitants c that have a close resemblance to the natural language hint h provided in the sketch. Hence, similar to the Table rule from Figure 4.7, we assign a confidence score by computing the similarity between c and h using Word2Vec.

Since most of the rules in Figure 4.8 are quite similar to the ones from Figure 4.7, we do not explain them in detail. However, we would like to draw the reader's attention to the *Pred* rule for instantiating predicate sketches of the form $?h\ op\ \eta$. Recall that a predicate $c\ op\ v$ evaluates to true for exactly those values v' in column c for which the predicate $v'\ op\ v$ is true. Now, if c does not contain any entries v' for which the $v'\ op\ v$ evaluates to true, there is a low, albeit non-zero, probability that $c\ op\ v$ is the intended predicate.

$$\frac{(c,\beta)\in\tau \quad p = sim(h,c)}{\Gamma,\tau\vdash_s ?h \Downarrow c:\{(c,\beta)\},\ p}\ \text{(Col)} \qquad \frac{p = P_\tau(v,\beta),\quad (c,\beta)\in\tau \quad cast(v,\beta)=v'}{\Gamma,\tau\vdash_s v\Downarrow v':\{(v',\beta)\},\ p}\ \text{(Value)}$$

$$\frac{\Gamma\vdash\chi\Downarrow T:\tau,p}{\Gamma,\tau\vdash_s\chi\Downarrow T:\tau,p}\ \text{(Reduce)} \qquad \frac{\Gamma,\tau\vdash_s\psi\Downarrow\phi:\mathbb{B},\ p}{\Gamma,\tau\vdash_s\neg\psi\Downarrow\neg\phi:\mathbb{B},\ p}\ \text{(PredNeg)}$$

$$\frac{\Gamma,\tau\vdash_s ?h\Downarrow c:\{(c,\beta)\},\ p_1 \quad \Gamma,\tau\vdash_s\eta\Downarrow E:\{(c',\beta)\},\ p_2 \quad p = p_1\otimes p_2\otimes P_\phi(c\ op\ E)}{\Gamma,\tau\vdash_s ?h\ op\ \eta\Downarrow c\ op\ E:\mathbb{B},\ p}\ \text{(Pred)}$$

$$\frac{\Gamma,\tau\vdash_s ?h\Downarrow c:\{(c,\beta)\},\ p \quad type(f)=\beta\to\beta'}{\Gamma,\tau\vdash_s f(?h)\Downarrow f(c):\{(f(c),\beta')\},\ p}\ \text{(Fun)}$$

$$\frac{\Gamma,\tau\vdash_s f(?h_1)\Downarrow f(c_1):\{(f(c_1),\beta)\},\ p_1 \quad \Gamma,\tau\vdash_s ?h_2\Downarrow c_2:\{(c_2,\tau_2)\},\ p_2}{\Gamma,\tau\vdash_s \quad g(f(?h_1),?h_2)\Downarrow g(f(c_1),c_2) : \{(c_2,\tau_2),(f(c_1),\beta)\},\ p_1\otimes p_2}\ \text{(Group)}$$

$$\frac{\Gamma,\tau\vdash_s\kappa_1\Downarrow L_1:\tau_1,\ p_1 \quad \Gamma,\tau\vdash_s\kappa_2\Downarrow L_2:\tau_2,\ p_2}{\Gamma,\tau\vdash_s\kappa_1,\kappa_2\Downarrow L_1,L_2:\tau_1\cup\tau_2,\ p_1\otimes p_2}\ \text{(ColList)}$$

$$\frac{\Gamma,\tau\vdash_s\psi_1\Downarrow\phi_1:\mathbb{B},\ p_1 \quad \Gamma,\tau\vdash_s\psi_2\Downarrow\phi_2:\mathbb{B},\ p_2}{\Gamma,\tau\vdash_s\psi_1\ lop\ \psi_2\Downarrow\phi_1\ lop\ \phi_2:\mathbb{B},\ p_1\otimes p_2}\ \text{(PredLop)}$$

Figure 4.8: Inference rules for specifiers

To capture this intuition, the *Pred* rule uses the following P_ϕ function when assigning a confidence score:

$$P_\phi(c \ op \ E) = \begin{cases} 1 - \epsilon & \text{if } \exists v' \in \mathit{contents}(c).\ v' \ op \ E = \top \\ \epsilon & \text{otherwise} \end{cases}$$

Here, ϵ is a small, non-zero constant that indicates low confidence. Hence, predicate $c \ op \ E$ is assigned a low score if there are no entries satisfying it in the database. [4]

Finally, we would also like to draw the reader's attention to the *Value* rule, which types constants in a quantitative manner. To gain intuition about the necessity of quantitatively typing constants, consider the string constant "forty two". While this value may correspond to a string constant, it could also be an integer (42) or a float (42.0). To deal with such ambiguity, our typing rules use a function P_τ to estimate the probability that constant v has type β and then cast v to a constant v' of type β.

Example 4.2. *Consider again the (tiny) database shown in Figure 4.4 and the following query sketch:*

$$\Pi_{\text{g(avg(?score), ?department)}}(??\text{score}_? \bowtie_? ??)$$

According to the rules from Figures 4.7 and 4.8, the query $\Pi_{\text{g(avg(score), dept)}}(\text{Grades}_{cid\_fk} \bowtie_{cid\_fk} \text{Grades})$ *is not a valid completion because it*

[4] Observe that this heuristic requires querying the underlying database. Hence, if SQLIZER is used as part of an online system that has direct access to the live database, our synthesis algorithm may place a load on the database by issuing multiple queries in short succession. However, this problem can be avoided by forking the database rather than using the live version.

is not "well-typed" (i.e., `dept` *is not a column in* $\text{Grades}_{cid\_fk} \bowtie_{cid\_fk} \text{Grades}$*). In contrast, the query* $\Pi_{\text{g(avg(score), dept)}}(\text{Grades}_{id} \bowtie_{cid} \text{Courses})$ *is well-typed but is assigned a low score because the column* `id` *in Grades is not a foreign key referring to column* `cid` *in Courses. As a final example, consider the query:*

$$\Pi_{\text{g(avg(score), dept)}}(\text{Grades}_{cid\_fk} \bowtie_{cid} \text{Courses})$$

This query is both well-typed and is assigned a high score close to 1.

4.7 Sketch Refinement Using Repair

In the previous section, we saw how to generate a ranked list of possible sketch completions using quantitative type inhabitation. However, in some cases, it may not be possible to find well-typed, high-confidence completions of *any* sketch. For instance, this situation can arise for at least two different reasons:

1. Due to ambiguities in the user's natural language description, the correct sketch may not be in the top k sketches generated by the semantic parser.

2. In some cases, the user's natural language description may be misleading. For instance, in the context of query synthesis, the user might make incorrect assumptions about the underlying data organization , so her English description may not accurately reflect the general structure of the target query.

Algorithm 4.2 Fault Localization Algorithm

```
1:  procedure FaultLocalize(𝔖, Γ, τ)
2:      Input: partial sketch 𝔖, schema Γ, record type τ
3:      Output: faulty partial sketch 𝔖′ or null
4:      if isRelation(𝔖) then
5:          for all (χ_i, ω_i) ∈ SubRelations(𝔖) do
6:              χ′_i = FaultLocalize(χ_i, Γ, τ)
7:              if χ′_i ≠ null then return χ′_i
8:              θ_i = FindInhabitants(χ_i, Γ)
9:              for all (ℑ_j, 𝔓_j, τ_j) ∈ θ_i do
10:                 ω′_ij = FaultLocalize(ω_i, Γ, τ_j)
11:             if ∀j. ω′_ij ≠ null then
12:                 if ∀j, k. ω′_ij = ω′_ik then return ω′_i0
13:                 else if CanRepair(ω_i) then return ω_i
14:     else if isSpecifier(𝔖) then
15:         for all ω_i ∈SubSpecifiers(𝔖) do
16:             ω′_i = FaultLocalize(ω_i, Γ, τ)
17:             if ω′_i ≠ null then return ω′_i
18:     θ = FindInhabitants(𝔖, Γ, τ)
19:     if MaxProb(θ) < ρ and CanRepair(𝔖) then
20:         return 𝔖
21:     else return null
```

One of the key ideas underlying our synthesis methodology is to overcome these problems through the use of *automated sketch refinement*. Given a faulty sketch $\mathfrak{S}$, the goal of sketch refinement is to generate a new program sketch $\mathfrak{S}'$ such that $\mathfrak{S}'$ repairs a potentially faulty sub-part of $\mathfrak{S}$. Similar to

$$
\begin{aligned}
\textsc{SubRelations}(\Pi_\kappa(\chi)) &= \{(\chi, \kappa)\} \\
\textsc{SubRelations}(\sigma_\psi(\chi)) &= \{(\chi, \psi)\} \\
\textsc{SubRelations}(\chi_1 {}_{?h_1}\!\bowtie_{?h_2} \chi_2) &= \{(\chi_1, ?h_1), (\chi_2, ?h_2)\} \\
\textsc{SubSpecifiers}(g(f(?h_1), ?h_2)) &= \{f(?h_1), ?h_2\} \\
\textsc{SubSpecifiers}(\kappa_1, \kappa_2) &= \{\kappa_1, \kappa_2\} \\
\textsc{SubSpecifiers}(?h \ op \ \eta) &= \{?h, \eta\} \\
\textsc{SubSpecifiers}(\neg\psi) &= \{\psi\} \\
\textsc{SubSpecifiers}(\psi_1 \ lop \ \psi_2) &= \{\psi_1, \psi_2\}
\end{aligned}
$$

Figure 4.9: Auxiliary functions used in Algorithm 4.2

prior approaches on automated program repair, our method performs sketch refinement using a combination of *fault localization* and a database of *repair tactics*. However, since we do not have access to a concrete test case that exhibits a specific bug, our sketch refinement procedure is again confidence-driven. Specifically, we perform fault localization by identifying a *minimal fault subpart* $\mathcal{F}$ of the sketch such that $\mathcal{F}$ does not have any high-confidence inhabitants. The *minimal faulty sub-sketch* $\mathcal{F}$ has the property that all of its strict sub-expressions have an inhabitant whose score exceeds our confidence threshold, but $\mathcal{F}$ itself does not.

4.7.1 Fault Localization

Algorithm 4.2 presents our fault localization algorithm for query sketches. The recursive FaultLocalize procedure takes as input the database schema Γ, a partial sketch $\mathcal{S}$, which can be either a relation or a specifier. If $\mathcal{S}$ is a specifier, FaultLocalize also takes as input a record type τ, which describes the schema for the parent table. (Recall that sketch completion for specifiers requires the parent table τ.) The return value of FaultLocalize is either the faulty sub-sketch or null (if $\mathcal{S}$ cannot be repaired).

Let us now consider Algorithm 4.2 in more detail. If $\mathcal{S}$ is a relation, we first recurse down to its subrelations and specifiers (see Figure 4.9) to identify a smaller subterm that can be repaired (lines 4–13). On the other hand, if $\mathcal{S}$ is a specifier (lines 14–17), we then recurse down to its subspecifiers, again to identify a smaller problematic subterm. If we cannot identify any such subterm, we then consider the current partial sketch $\mathcal{S}$ as the possible cause of failure. That is, if $\mathcal{S}$ does not have any inhabitants that meet our confidence threshold, we then check if $\mathcal{S}$ can be repaired using our database of repair tactics. If so, we return $\mathcal{S}$ as the problematic subterm (lines 18–20).

One subtle part of the fault localization procedure is the handling of specifiers in lines 11–13. Recall from Section 4.6 that the completion of specifiers is dependent on the parent relation. Specifically, when we find the inhabitant of a relation such as $\Pi_\kappa(\chi)$, we need to know the type of χ when we complete κ. Hence, there is a valid completion of $\Pi_\kappa(\chi)$ if there *exists* a valid completion of κ for *some* inhabitant of χ. Thus, we can only say that ω_i (or

one of its subterms) is faulty if it is faulty for all possible inhabitants of χ (i.e., $\forall j.\ \omega'_{ij} \neq \text{null}$).

Example 4.3. *Consider the query "Find the number of papers in OOPSLA 2010" from the motivating example in Section 4.2. The initial sketch generated by semantic parsing is:*

$$\Pi_{\text{count(?papers)}}(\sigma_{\text{?=OOPSLA 2010}}(\text{??papers}))$$

Since there is no high-confidence completion of this sketch, we perform fault localization using Algorithm 4.2. The innermost hole `??papers` *can be instantiated with high confidence, so we next consider the subterm* `? = OOPSLA 2010`*. Observe that there is no completion of* `??papers` *under which* `? = OOPSLA 2010` *has a high confidence score because no table in the database contains the entry* `"OOPSLA 2010"`*. Hence, fault localization identifies the predicate as* `? = OOPSLA 2010` *as the root cause of failure.*

4.7.2 Repair Tactics

Once we identify a faulty subpart $\mathcal{F}$ of sketch $\mathcal{S}$, our method tries to repair $\mathcal{S}$ by replacing $\mathcal{F}$ by some $\mathcal{F}'$ obtained using a database of domain-specific repair tactics. Figure 4.10 shows a representative subset of our repair tactics for the query synthesis domain. At a high-level, our repair tactics describe how to introduce new predicates, join operators, and columns into the relevant sub-parts of the sketch.

To gain some intuition about our repair tactics for database queries, let

$$\frac{split(v) = (v_1, v_2), \quad v_2 \neq \epsilon}{?h\ op\ v \rightsquigarrow ?h\ op\ v_1 \wedge ?h\ op\ v_2} \qquad \text{(AddPred)}$$

$$\frac{}{\sigma_\psi(\chi) \rightsquigarrow \sigma_\psi(\chi_{?\epsilon} \bowtie_{?\epsilon} ??\epsilon)} \qquad \text{(AddJoin1)}$$

$$\frac{}{\Pi_\kappa(\chi) \rightsquigarrow \Pi_\kappa(\chi_{?\epsilon} \bowtie_{?\epsilon} ??\epsilon)} \qquad \text{(AddJoin2)}$$

$$\frac{}{\chi_{1\,?h_1} \bowtie_{?h_2} \chi_2 \rightsquigarrow \chi_{1\,?\epsilon} \bowtie_{?\epsilon} ??\epsilon_{?\epsilon} \bowtie_{?\epsilon} \chi_2} \qquad \text{(AddJoin3)}$$

$$\frac{\begin{array}{c} split(h) = (f', h') \\ f \in AggrFunc, \quad sim(f, f') \geq \delta \end{array}}{?h \rightsquigarrow f(?h')} \qquad \text{(AddFunc)}$$

$$\frac{}{?h\ op\ v \rightsquigarrow ?h\ op\ ?v} \qquad \text{(AddCol)}$$

Figure 4.10: Repair tactics. Here, $split(v)$ tokenizes value v using predefined delimiters: $split(v) = (v_1, v_2)$ iff v_1 occurs before the first occurrence of the delimiter and v_2 occurs after. If the delimiter doesn't appear in v, then $split(v) = (v, \epsilon)$.

us consider the rewrite rules in Figure 4.10 in more detail. The first tactic, labeled *AddPred*, splits a predicate into two parts by introducing a conjunct. For instance, consider a predicate $?h\ op\ v$ and suppose that v is a string that contains a common delimiter (e.g., space, apostrophe etc.). In this case, the *AddPred* tactic splits v into two parts v_1, v_2 occurring before and after the delimiter and rewrites the predicate as $?h\ op\ v_1 \wedge ?h\ op\ v_2$. For example, we

have used this tactic in the motivating example from Section 4.2 when rewriting `?="OOPSLA 2010"` as `?="OOPSLA"` and `?="2010"`.

The rewrite rules labeled *AddJoin* in Figure 4.10 show how to introduce additional join operators in selections, projections, and joins. Because users may not be aware that the relevant information is spread across multiple database tables, these tactics allow us to introduce join operators when the user's English description does not contain any clues about the necessity of performing joins. For instance, recall that we use the *AddJoin* tactic in our example from Section 4.2 to rewrite the term `??[papers]` into `??[papers] JOIN ??`.

The next rule labeled *AddFunc* introduces an aggregate function if the hint h in $?h$ contains the name of an aggregate function (e.g., count) or something similar to it. For instance, consider a hole `?` with hint *"average grade"*. Since *"average"* is also an aggregate function, the *AddFunc* rule can be used to rewrite this as *avg(?grade)*. Finally, the last rule labeled *AddCol* introduces a new column in the predicate $?h\ op\ v$. Since v may refer to the name of a column rather than a constant string value, the *AddCol* rule allows us to consider this alternative possibility.

As mentioned earlier, the particular repair tactics used in the context of sketch refinement are quite domain-specific. However, since natural language is inherently imprecise and ambiguous, we believe that the proposed methodology of refining the program sketch using fault localization and repair tactics would also be beneficial in other contexts where the goal is to synthesize a program

from natural language.

4.8 Implementation

Our SQLIZER tool, written in a combination of C++ and Java, automatically synthesizes SQL code from English queries. SQLIZER uses the SEMPRE framework [29] and the Stanford CoreNLP library [124] in the implementation of its semantic parser. For quantitative type inhabitation, SQLIZER uses the Word2Vec [129] tool to compute a similarity metric between English hints in the sketch and names of database tables and columns.

Recall that our synthesis algorithm presented in Section 4.3 only considers the top k query sketches and repairs each program sketch at most n times. It also uses a threshold γ to reject low-confidence queries. While each of these parameters can be configured by the user, the default values for k and n are both 5, and the default value for γ is 0.35. In our experimental evaluation, we also use these default values.

4.8.1 Training Data

Recall from Section 4.5 that our semantic parser uses supervised machine learning to optimize the weights used in the likelihood score for each utterance. Towards this goal, we used queries for a mock database accompanying a databases textbook [54]. Specifically, this database contains information about the employees, departments, and projects for a hypothetical company. In order to train the semantic parser, we extracted the English descriptions of 108

queries from the textbook and manually wrote the corresponding sketch for each query. Please note that the query sketches were constructed directly from the English description *without manually "repairing" them to fit the actual database schema*. Also, while a training set of 108 queries may seem like a small number compared to other supervised machine learning techniques, we can get away with such a modest amount of training data for several reasons: First, due to our use of query sketches as the logical form representation, we do not need to train a statistical model to map English phrases to relational algebra expressions over the database schema. Second, the grammar we implemented in the semantic parser is carefully designed to minimize ambiguities. Finally, the linguistic processor used in the semantic parser is pre-trained over a large corpus, including ∼2,500 articles from the Wall Street Journal for part-of-speech tagging and ∼15,000 sentences from the CoNLL-2003 dataset for named entity recognition.

4.8.2 Optimizations

While our implementation closely follows the technical presentation in this chapter, it performs two important optimizations that we have not mentioned previously: First, because the fault localization procedure completes the same sub-sketch many times, we memoize the result of sketch completion for every subterm. Second, if the score for a subterm is less than a certain confidence threshold, we reject the partially completed sketch without trying to complete the remaining holes in the sketch.

4.9 Evaluation

To evaluate SQLIZER, we perform experiments that are designed to answer the following questions:

Q1. How effective is SQLIZER at synthesizing SQL queries from natural language descriptions?

Q2. What is SQLIZER's average running time per query?

Q3. How well does SQLIZER perform across different databases?

Q4. How does SQLIZER perform compared to other tools for synthesizing queries from natural language?

Q5. What is the relative importance of type information and sketch repair in practice?

Q6. How important are the various heuristics that we use to assign confidence scores to type inhabitants?

4.9.1 Experimental Setup

To answer these research questions, we evaluate SQLIZER on three real-world databases, namely the Microsoft academic search database (MAS) used for evaluating NALIR [113], the Yelp business reviewing database (YELP), and the IMDB movie database (IMDB). [5] Table 4.1 provides statistics about each database.

Database	Size	#Tables	#Columns
MAS	3.2GB	17	53
IMDB	1.3GB	16	65
YELP	2.0GB	7	38

Table 4.1: Database Statistics

Cat	Description
C1	Does not use aggregate function or join operator
C2	Does not use aggregate function but Joins different tables
C3	Uses an aggregate function
C4	Uses subquery or self-join

Table 4.2: Categorization of different benchmarks

Benchmarks. To evaluate SQLIZER on these databases, we collected a total of 455 natural language queries. For the MAS database, we use exactly the 196 benchmarks obtained from the NALIR dataset [113]. For the IMDB and YELP databases, we asked a group of people at our organization to come up with English queries that they might like to answer using IMDB and YELP. Participants were given information about the type of data available in each database (e.g., business names, cities, etc.), but they did not have any prior knowledge about the underlying database schema, including names of database tables and columns.

Checking correctness. To evaluate the accuracy of SQLIZER, we manually inspected the SQL queries returned by SQLIZER. We consider a query to be

[5]All the benchmarks and databases are available at goo.gl/DbUBMM

correct if (a) executing the query yields the desired information, and (b) the synthesized query faithfully implements the data retrieval logic specified by the user's English description.

Categorization of benchmarks. To assess how SQLIZER performs on different classes of queries, we manually categorize the benchmarks into four groups based on the characteristics of their corresponding SQL query. Table 4.2 shows our taxonomy and provides an English description for each category. While there is no universal agreement on the difficulty level of a given database query, we believe that benchmarks in category C_{i+1} are generally harder for humans to write than benchmarks in category C_i.

Hardware and OS. All of our experiments are conducted on an Intel Xeon(R) computer with E5-1620 v3 CPU and 32GB memory, running the Ubuntu 14.04 operating system.

4.9.2 Accuracy and Running Time

Table 4.3 summarizes the results of evaluating SQLIZER on the 455 benchmarks involving the MAS, IMDB, and YELP databases. In this table, the column labeled "Count" shows the number of benchmarks under each category. The columns labeled "Top k" show the number (#) and percentage (%) of benchmarks whose target query is ranked within the top k queries synthesized by SQLIZER. Finally, the columns labeled "Parse time" and "Synth/repair time" show the average time (in seconds) for semantic parsing and sketch

DB	Cat	Count	Top 1		Top 3		Top 5		Parse time (s)	Synth/ repair time(s)
			#	%	#	%	#	%		
MAS	C1	14	12	85.7	14	100.0	14	100.0	0.41	0.08
	C2	59	52	88.1	55	93.2	55	93.2	1.07	0.16
	C3	60	49	81.7	55	91.6	56	93.3	1.11	0.29
	C4	63	45	71.4	49	77.7	53	84.1	2.53	0.21
	Total	**196**	**158**	**80.6**	**173**	**88.3**	**178**	**90.8**	**1.50**	**0.21**
IMDB	C1	18	16	88.9	17	94.4	17	94.4	0.50	0.21
	C2	69	51	73.9	59	85.5	61	88.4	0.60	0.24
	C3	27	24	88.8	26	96.2	26	96.2	0.71	0.34
	C4	17	11	64.7	11	64.7	12	70.5	0.70	0.49
	Total	**131**	**102**	**77.9**	**113**	**86.3**	**116**	**88.5**	**0.61**	**0.28**
YELP	C1	8	6	75.0	7	87.5	7	87.5	0.55	0.02
	C2	49	35	71.4	39	79.6	42	85.7	0.77	0.05
	C3	51	40	78.4	48	94.1	49	96.0	0.72	0.05
	C4	20	15	75.0	15	75.0	15	75.0	0.96	0.06
	Total	**128**	**96**	**75.0**	**109**	**85.2**	**113**	**88.3**	**0.77**	**0.05**

Table 4.3: Summary of Sqlizer's experimental evaluation

completion/refinement respectively.

As shown in Table 4.3, Sqlizer achieves close to 90% accuracy across all three databases when we consider a benchmark to be successful if the desired query appears within the top 5 results. Even if we adopt a stricter definition of success and consider Sqlizer to be successful if the target query is ranked within the top one (resp. top three) results, Sqlizer still achieves approximately 78% (resp. 86%) accuracy. Also, observe that Sqlizer's

synthesis time is quite reasonable; on average, SQLIZER takes 1.22 seconds to synthesize each query, with 85% of synthesis time dominated by semantic parsing.

To gain intuition about cases in which SQLIZER does not work well, we investigated the root causes of failures in our experimental evaluation. In most cases, the problem is caused by domain-specific terms for which we cannot accurately compute similarities using Word2Vec. For instance, in the context of on-line reviewing systems such as Yelp, the terms "star" and "rating" are used interchangeably, but the domain-agnostic Word2Vec system does not consider these two terms to be similar. Clearly, this problem can be alleviated by training the neural net for measuring word similarity on a corpus specialized for this domain. However, since our goal is to develop a database-agnostic system, we have not performed such domain-specific training for our evaluation.

4.9.3 Comparison with NALIR

To evaluate whether the results described in Section 4.9.2 improve over the state-of-the-art, we also compare SQLIZER against NALIR, a recent system that won a best paper award at VLDB'14 [113]. Rather than re-implementing the ideas proposed in the VLDB'14 paper, we directly use the NALIR implementation provided to us by NALIR's developers.

Similar to SQLIZER, NALIR generates SQL queries from English and also aims to be database-agnostic (i.e., does not require database-specific training). However, unlike SQLIZER, which is fully automated, NALIR can also be used

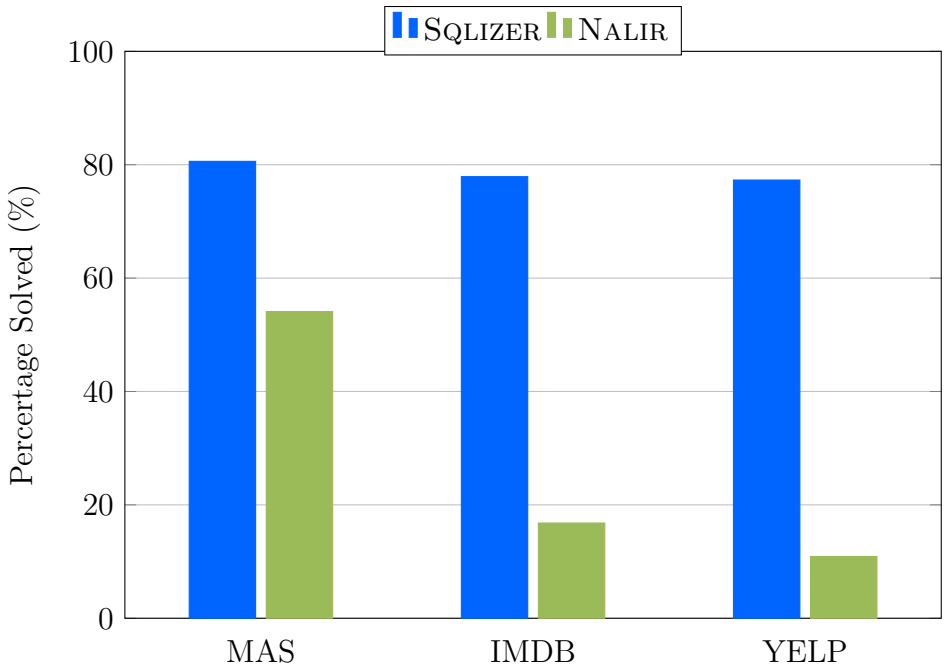


Figure 4.11: Comparison between SQLIZER and NALIR

in an interactive setting that allows the user to provide guidance by choosing the right query structure or the names of database elements. In order to perform a fair comparison between SQLIZER and NALIR, we use NALIR in the non-interactive setting (recall that SQLIZER is fully automatic). Furthermore, since NALIR only generates a single database query as its output, we compare NALIR's results with the *top-ranked query* produced by SQLIZER.

As shown in Figure 4.11, SQLIZER outperforms NALIR on all three databases with respect to the number of benchmarks that can be solved. In particular, SQLIZER's average accuracy is 78% whereas NALIR's average accuracy is less than 32%. Observe that the queries for the MAS database are

the same ones used for evaluating NALIR, but SQLIZER outperforms NALIR even on this dataset. Furthermore, even though SQLIZER's accuracy is roughly the same across all three databases, NALIR performs significantly worse on the IMDB and YELP databases.

To provide some intuition about why SQLIZER performs better than NALIR in our experiments, recall that SQLIZER can automatically refine query sketches using a database of repair tactics and guided by the confidence scores inferred during sketch completion. In contrast, NALIR does not automatically resolve ambiguities and is more effective when it is used in its interactive mode that allows the user to guide the system by choosing between different candidate mappings between nodes in the parse tree to SQL components.

4.9.4 Evaluation of Different Components of Synthesis Methodology

In this chapter, we argued that the use type information and automatic sketch refinement are both very important for effective synthesis from natural language. To justify this argument, we compare SQLIZER against two variants of itself. One variant, referred to as **NoType**, does not use type information to reject some of the generated SQL queries. The second variant, referred to as **NoRepair**, does not perform sketch refinement.

Figure 4.12 shows the results of our evaluation comparing SQLIZER against these two variants of itself. As we can see from this figure, disabling either of these features dramatically reduces the accuracy of the system. In

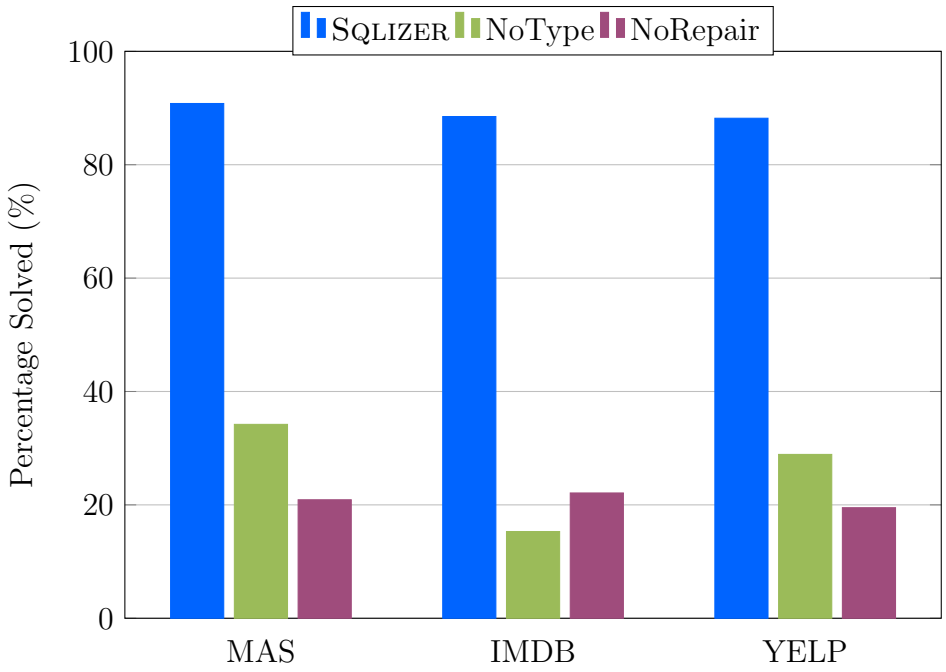


Figure 4.12: Comparison between different variations of SQLIZER on Top 5 results

particular, while the full SQLIZER system ranks the target query within the top 5 results in over 88% of the cases, the average accuracy of both variants is below 35%. We believe that these results demonstrate that the use of types and automated repair are both crucial for the overall effectiveness of SQLIZER.

4.9.5 Evaluation of Heuristics for Assigning Confidence Scores

A key ingredient of the synthesis methodology is *quantitative type inhabitation* in which we use domain-specific heuristics to assign confidence scores to programs. In the context of the database domain, we proposed three different heuristics for assigning confidence scores to queries. The first heuristic,

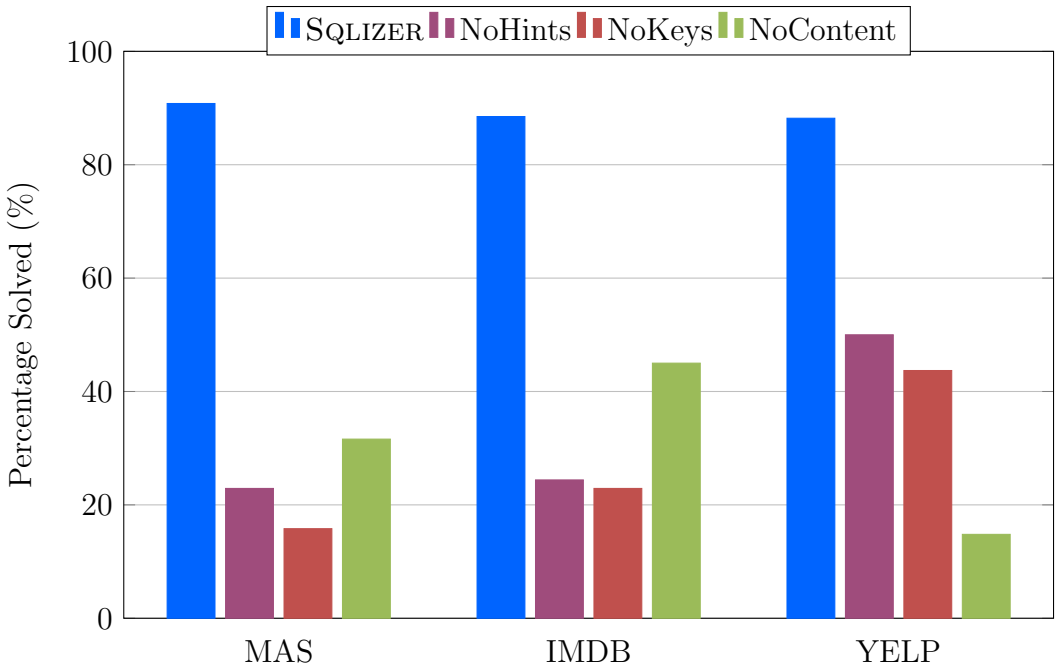


Figure 4.13: Impact of heuristics for assigning confidence scores on Top 5 results

referred to as *Hints*, computes the similarity between the natural language hints in the sketch and the names of schema elements. The second heuristic, referred to as *Keys*, uses a function $P_{\bowtie}$ to assign scores to join operators using information about foreign keys. The third heuristic, referred to as *Content*, uses a function P_{ϕ} that assigns confidence scores to selection operations using the contents of the database.

We evaluate the relative importance of each of these heuristics in Figure 4.13. Specifically, the variant of SQLIZER labeled *NoX* disables the scoring heuristic called X. As we can see from the figure, all of our proposed heuristics are quite important for SQLIZER to work effectively as an end-to-end synthesis

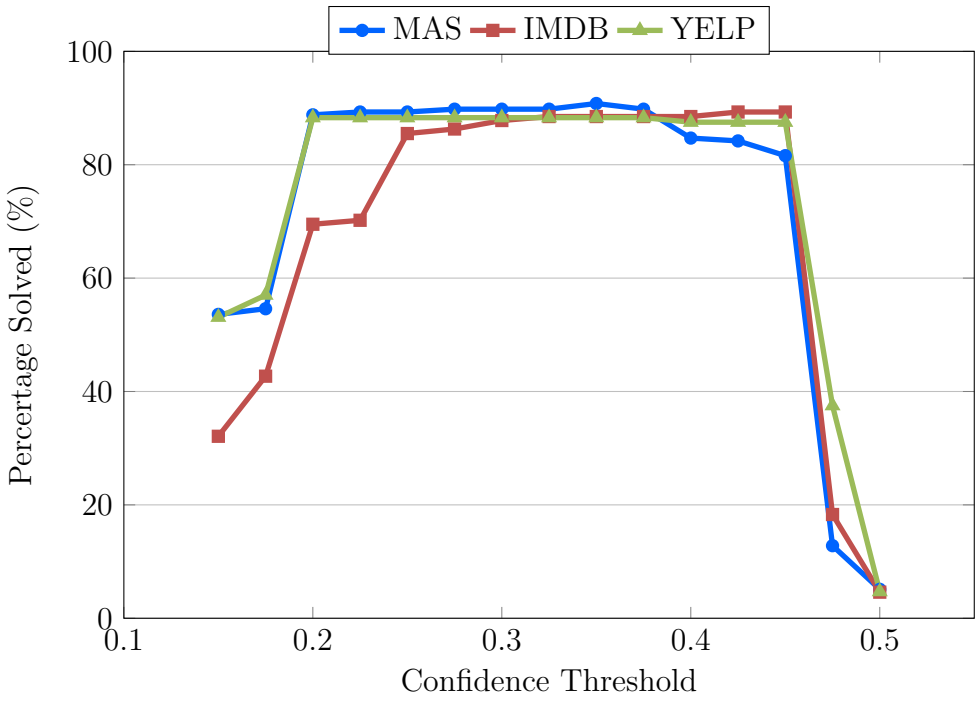


Figure 4.14: Impact of different confidence thresholds on Top 5 results

tool.

4.9.6 Evaluation of Different Confidence Thresholds

Recall from Algorithm 4.1 that our synthesis algorithm rejects queries with a confidence score below a certain threshold γ. As mentioned in Section 4.8, we use the default value $\gamma = 0.35$ in all of our experiments. To understand the impact of this confidence threshold on our results, we also run the same set of experiments using different confidence thresholds in the range $[0.15, 0.5]$. As shown in Figure 4.14, SQLIZER is not very sensitive to the exact value of γ as long as it is in the range $[0.25, 0.45]$; however, accuracy drops sharply if γ

is chosen to be either below 0.25 or above 0.45. If the threshold γ is too low (i.e., below 0.25), SQLIZER seems to generate many incorrect queries, thereby affecting the overall ranking of the target query. On the other hand, if γ is too high (i.e., above 0.45), SQLIZER ends up ruling out more queries, sometimes including the desired query. Hence, overall precision decreases in both cases.

4.10 Limitation

In this section, we discuss the current limitations of our system and possible ways to improve it in the future.

Recall that SQLIZER uses domain-specific heuristics to assign confidence scores to query completions, and one of these heuristics (namely, P_ϕ) uses database contents when performing quantitative type inhabitation. The strategy of assigning low confidence scores to queries that yield an empty relation works very well in most cases, but it may prevent SQLIZER from generating the right query if the query legitimately returns an empty table. Another situation in which this heuristic may not work well is if the user's natural language query does not *exactly* match the contents of the database, such as in cases where the user's description uses an abbreviation or contains a misspelling. A possible solution to this problem is to look for syntactically or semantically similar entries in the database rather than insisting on an exact match.

Another limitation of SQLIZER is that the user ultimately needs to decide which (if any) of the top k queries returned by SQLIZER is the right one. An interesting avenue for future work is to explore user-friendly mechanisms to

help the user make this decision. One possibility is to present the user with a natural language description of the query rather than the SQL query itself. However, this solution would still require the user to be knowledgeable about the underlying database schema. Another possibility is to present the query *result* rather than the query itself and let the user decide if the result is sensible. An even better solution would be to further improve the system's accuracy so that the desired SQL query is ranked number one in more cases. However, given that SQLIZER is already capable of returning the desired query as the top result in 78% of the cases, we believe that SQLIZER is still quite useful to end-users as is.

4.11 Summary

In this chapter, we have proposed a new methodology for synthesizing programs from natural language and applied it to the problem of synthesizing SQL code from English queries. Starting with an initial program sketch generated using semantic parsing, our approach enters an iterative refinement loop that alternates between quantitative type inhabitation and sketch repair. Specifically, our method uses domain-specific knowledge to assign confidence scores to type (i.e., sketch) inhabitants and uses these confidence scores to guide fault localization. The faulty subterms pinpointed using error localization are then repaired using a database of domain-specific repair tactics.

We have implemented the proposed approach in a tool called SQLIZER, an end-to-end system for generating SQL queries from natural language. Our

experiments on 455 queries from three different databases shows that SQLIZER ranks the desired query as top one in 78% of the cases and among top 5 in $\sim$ 90% of the time. Our experiments also show that SQLIZER significantly outperforms NALIR, a state-of-the-art system for generating SQL code from natural language queries. Finally, our evaluation also justifies the importance of type information and program repair and shows that our proposed domain-specific heuristics are necessary and synergistic.

Chapter 5

Related Work

The work presented in this dissertation is related to a variety of different topics from the programming languages, databases, and natural language processing communities. In what follows, we compare and our work against existing techniques.

5.1 Program Synthesis

Program synthesis has recently received much attention in the programming languages community. The techniques proposed in this dissertation borrow insights from other researches on program synthesis. In particular, the use of the term *sketch* in Sqlizer is inspired by the Sketch system [165, 166, 164] in which the user writes a *program sketch* containing holes (unknown expressions). However, in contrast to the Sketch system where the holes are instantiated with constants, holes in our query sketches are completed using database tables, columns, and predicates. Similar to Sqlizer, some prior techniques (e.g., [194, 60]) have also decomposed the synthesis task into two separate sketch generation and sketch completion phases. However, to the best of our knowledge, we are the first to generate program sketches from natural language

using semantic parsing.

In addition to program sketching, our SQL query synthesizer also draws insights from recent work on *type-directed program synthesis* [135, 80, 140, 62]. Among these projects, the most related one is the InSynth system, which synthesizes small code snippets in a type-directed manner. Similar to our proposed methodology in Chapter 4, InSynth also performs quantitative type inhabitation to synthesize type-correct expressions at a given program point. Specifically, InSynth assigns weights to each type inhabitant, where lower weights indicate higher relevance of the synthesized term. These weights are derived using a training corpus and the structure of the code snippet. At a high level, our use of quantitative type inhabitation is similar to InSynth in that we both use numerical scores to evaluate which term is most likely to be the inhabitant desired by the user. However, the way in which we assign confidence scores to type inhabitants is very different from the way InSynth assigns weights to synthesized code snippets. Furthermore, in contrast to InSynth where lower weights indicate higher relevance, Sqlizer assigns lower scores to inhabitants that are less likely to be correct.

5.1.1 Programing by Natural Language (PBNL)

The techniques we proposed in Chapter 4 are also related to *programming by natural language* [78, 47, 149, 148, 110]. Natural language has been used as the preferred specification mechanism in various contexts such as smartphone automation scripts [110], "if-then-else recipes" [148], spreadsheet

programming [78], and string manipulation [149]. Among these systems, the NLyze tool [78] is most closely related to Sqlizer in that it also combines semantic parsing with type-directed synthesis. However, NLyze does not generate program sketches and uses type-based synthesis to mitigate the low recall of semantic parsing. In contrast, Sqlizer uses semantic parsing to generate an initial query sketch, which is refined using repair tactics and completed using quantitative type inhabitation. Furthermore, NLyze targets spreadsheets rather than relational databases and proposes a new DSL for this domain.

Most recently, [47] have proposed a general framework for constructing synthesizers that can generate programs in a domain-specific DSL from English descriptions. The framework requires a DSL definition and a set of domain-specific training data in the form of pairs of English sentences and their corresponding programs in the given DSL. While this framework can, in principle, be instantiated for SQL query synthesis, it would require database-specific training data.

5.1.2 Programing by Example (PBE)

The proposed methods in Chapters 2 and 3 perform synthesis from examples. The problem of automatically synthesizing programs that satisfy a given set of input-output examples has been the subject of research in the past four decades [159]. Recent advances in algorithmic and logical reasoning techniques have led to the development of PBE systems in a variety of domains including string transformations [73, 161], data filtering [178], data structure

manipulations [61], table transformations [59, 83], SQL queries [176, 194] network policies [190], and map-reduce distributed programs [163].

Many of these approaches focus on non-hierarchical data like numbers, strings, and tables [162, 74, 84, 27, 138]. However, several recent efforts study the synthesis of programs over recursive data structures [98, 13, 62, 136, 109]. For example, FlashExtract synthesizes programs made from higher-order combinators using a custom deductive procedure [109], and Escher uses goal-directed enumerative search to synthesize first-order programs with recursive functions [13]. Similarly, λ^2 synthesizes higher-order functional programs using a combination of deduction and cost-directed enumeration [62]. Osera et al. study a similar problem and offer a solution based on type-directed deduction [136].

Comparison with HADES**.** Unlike the above approaches, HADES algorithm is based on a reduction of synthesis over trees to synthesis over lists, which is performed using SMT solving and decision tree learning. Furthermore, since in HADES we specifically target hierarchical data transformations (rather than synthesis of arbitrary programs), this tighter focus allows us handle tree transformation benchmarks whose complexity exceeds those in prior work. For example, unlike prior efforts, our method is able to synthesize transformations that alter the hierarchical structure of an input tree. So far as we know (and as demonstrated empirically in Section 2.7), such benchmarks fall outside the scope of prior approaches to example-driven synthesis.

Our subroutine for synthesizing path transformations in HADES bears similarities to FlashFill's strategy for synthesizing string transformations [74]. Similar to our INFERPATHTRANS procedure, FlashFill uses a combination of partitioning and unification; but the algorithmic details are very different. For example, FlashFill maintains a DAG representation of all string transformations that fit a set of examples. In contrast, we use a numerical representation of the input-output examples and reduce unification to SMT solving.

The StriSynth tool described in [79] extends FlashFill and uses it to automate certain kinds of file manipulation tasks, such as renaming files and directories. While the StriSynth approach can handle sophisticated transformations involving file names, it does not address transformations that handle the directory structure. In contrast, HADES addresses general tree transformations and can synthesize bash scripts that modify the directory structure.

Comparison with MITRA. Among existing PBE techniques, HADES is the most relevant work to MITRA. While HADES uses a similar internal representation as HDTs defined in MITRA, HADES focuses on tree-to-tree transformations and cannot automate transformations from hierarchical to relational data. Although a relational table can be represented as an HDT, HADES' approach of decomposing trees to a set of paths omits relations between different paths (i.e., columns in the relational table) and can therefore not automate most kinds of interesting tree-to-table transformations. To the best of our knowledge, the only prior PBE-based tool that aims to automate transformations from semi-structured to relational data is FLASHEXTRACT [108]. While the main

focus of FlashExtract is data stored in spreadsheets, it also supports some tree-structured data, such as HTML documents. However, FlashExtract is less expressive than Mitra, as it cannot infer relations between different nodes in the tree structure.

5.2 Databases

Out proposed techniques are also related to recent work in the database community. In particular, our methodology for automatic migration of hierarchical data to relational tables is related to *data exchange* and *XML to relational mapping* researches, and the idea of *query synthesis* has been studied in depth in the database community.

5.2.1 Data Exchange

The problem of converting hierarchically structured documents to a relational format is a form of *data exchange* problem, where the goal is to transform a data instance of a source schema into a data instance of a target schema [57]. Due to the difficulty of manually performing such transformations, there has been significant work on automating data exchange tasks [142, 56, 150, 130, 58]. A common approach popularized by *Clio* [142] decomposes the data exchange task into two separate phases: *schema mapping* and *program generation*. A schema mapping describes the relationship between the source and target schemas and is typically specified in the form of declarative logic constraints, such as GLAV (Global-and-Local-As-View) constraints [102]. Given

a schema mapping, the second program generation phase "translates" this mapping into executable code [56].

Because manual construction of schema mappings requires non-trivial effort from data architects, there has been significant work on automatically inferring such mappings from various kinds of informal specifications provided by the user. For instance, users may specify element correspondences by drawing lines between elements that contain related data [56, 94, 122, 50, 132, 55]. More recently, examples have become more popular as a way of communication between users and the system. Such example-based approaches can be broadly categorized into two classes depending on what the examples are used for. One approach uses examples to specify the desired schema mapping [145, 16, 14, 17]. To the best of our knowledge, all of these approaches use GLAV (or similar formalisms) as the schema mapping language, and hence can only handle cases where the source and target schemas are both relational. The other approach uses examples to help users understand and refine the generated schema mappings [187, 15], and the learning procedure still takes visual correspondences (lines between elements) as specification. Mitra can be viewed as an instantiation of the first approach. However, our method is different from previous methods of this approach in three important aspects. First, we can synthesize programs that convert tree-structured to relational data. Second, we design a DSL as the intermediate language which can express mappings between hierarchical and relational formats. Finally, our PBE-based synthesis algorithm combines finite automata and predicate learning to enable

efficient mapping generation.

5.2.2 XML-to-Relational Mapping

There is a significant body of research on using relational database management systems to store and query XML (and JSON) documents [103, 155, 52, 188, 88, 172, 156, 20, 24]. A typical approach for this problem consists of three steps: First, the tree structure of the document is converted into a flat, relational schema; next, the XML document is "shredded" and loaded into the relational database tables, and, finally, XML queries are translated into corresponding SQL queries. The goal of these systems is quite different from Mitra's: In particular, they aim to efficiently answer queries about the XML document by leveraging RDBMS systems, whereas our goal is to answer SQL queries on a desired relational representation of the underlying data.

5.2.3 Query Synthesis

There is a significant body of work on automatically synthesizing database queries. Related work in this area can be categorized into three classes, depending on the form of specifications provided by the user. In one line of work on query synthesis, users convey their intent to the system through the use of *input-output examples* [194, 173, 196, 176]. Specifically, the input to the system is a miniature version of the database, and the output is the desired table that should be extracted from this database using the target query. In our Sqlizer work, we prefer natural language specifications over input-output

examples for two important reasons: First, in order to provide input-output examples, the user must be familiar with the database schema, which is not always the case. Second, since a database may contain several tables with many different columns, providing input-output examples may be prohibitively cumbersome.

The second line of work on query synthesis uses *natural language descriptions* to convey user intent [21, 22, 179, 114, 144, 143, 113]. Early work in this area focuses on systems that are hand-crafted to specific databases [183, 85, 179, 43]. Later work describes NLIDB systems that can be reused for multiple databases with appropriate customization [70, 192, 170]. However, these techniques are not database-agnostic in that they require additional customization for each database. In contrast, our technique in Chapter 4 does not require database-specific training.

Similar to our proposed approach in SQLIZER, the NALIR system [113] also aims to be database-agnostic. Specifically, NALIR leverages an English dependency parser to generate linguistic parse trees, which are subsequently translated into *query trees*, possibly with guidance from the user. In addition to being relatively easy to convert to SQL, these query trees can also be translated back into natural language with the goal of facilitating user interaction. In contrast, our goal in SQLIZER is to develop a system that is as reliable as NALIR without requiring guidance from the user. In particular, since users may not be familiar with the underlying database schema, it may be difficult for them to answer some of the questions posed by a NALIR-style system. In contrast,

our approach does not assume that users are familiar with the organization of information in the database.

The third line of work on query synthesis generates more efficient SQL queries from code written in conventional programming languages [181, 41]. For instance, QBS [41] transforms parts of the application logic into SQL queries by automatically inferring loop invariants. This line of work is not tailored towards end-users and can be viewed as a form of query optimization using static analysis of source code.

5.3 Natural Language Processing

Unlike syntactic parsing which focuses on the grammatical divisions of a sentence, semantic parsing aims to represent a sentence through logical forms expressed in some formal language [29, 116, 191, 97, 170, 106]. Previous techniques have used semantic parsing to directly translate English sentences to database queries [131, 192, 97, 170, 106]. Unlike these techniques, in SQLizer we only use semantic parsing to generate an initial query sketch rather than the full query. We believe that there are two key advantages to our approach: First, our technique can be used to answer queries on a database on which it has not been previously trained. Second, the use of sketch refinement allows us to handle situations where the user's description does not accurately reflect the underlying database schema.

5.4 Program Repair

The sketch refinement strategy in SQLIZER is inspired by prior work on *fault localization* [25, 68, 91, 90, 93, 92] and *program repair* [119, 120, 180, 67, 133]. Similar to other techniques on program repair, the repair tactics employed by SQLIZER can be viewed as a pre-defined set of templates for mutating the program. However, to the best of our knowledge, we are the first to apply repair at the level of program sketches rather than programs.

Fault localization techniques [25, 68, 91, 90, 93, 92] aim to pinpoint a program expression that corresponds to the root cause of a bug. Similar to these fault localization techniques, SQLIZER tries to identify the minimal faulty subpart of the "program". However, we perform fault localization at the level of program sketches by finding a minimal sub-sketch for which no high-confidence completion exists.

Chapter 6

Conclusion and Future Work

Lack of programming knowledge forces many end-users to spend a considerable amount of time on manually performing data manipulation tasks such as restructuring data structures or converting them from one format to another. To address this problem, in this dissertation we introduced new methodologies for automatic generation of data transformation and extraction programs from informal specifications. In particular, we presented a new system, HADES, for synthesizing hierarchical data transformation programs from input-output examples. We also introduced MITRA, an example based synthesis tool which automatically migrates hierarchical data structures to relational tables. Finally, we proposed a novel approach for generating SQL queries from natural language in a system called SQLIZER.

We evaluated all of our systems on real-world benchmarks collected from online forums such as *Stackoverflow* and other publicly available data sources (e.g. Microsoft Academic Search database). Our results demonstrated that our systems can efficiently synthesize programs to automate practical data manipulation tasks. Moreover, since all of these systems only rely on informal problem descriptions, they can be used by any end-users, regardless of their

programming knowledge.

We believe future research can improve the systems presented in this dissertation to overcome their current limitations. In particular, the algorithm presented in Chapter 2 for synthesizing tree to tree transformations can only synthesize tree transformations that are expressible as a combination of independent path transformations. That is, our method cannot synthesize a program where the transformation for one path of the tree depends on a property of a different path. While the majority of the tasks we have encountered on online forums conform to our requirements, not every tree transformation script is synthesizable using our approach. For future work, we are interested in extending our system to overcome this limitation and be able to synthesize a broader class of tree transformations. Also, we believe that the idea of reducing the synthesis of a tree transformation program to synthesizing a transformation program over its path can be used to automate the transformation of other data structures such as graphs.

In order to improve our methodology for migrating tree-structured documents to relational tables, we are interested in further optimizing the data migration programs generated by Mitra. In particular, our method performs data migration by running a separate program per database table. However, because these programs operate over the same dataset, we can reduce overall execution time by memoizing results across different programs. Another limitation is that our synthesis algorithm described in Chapter 3 does not return anything if there is no DSL program that satisfies the given examples. To

provide more meaningful results to the user in such cases, we plan to investigate techniques that can synthesize programs that maximize the number of satisfied input-output examples.

In the future, we also plan to apply our proposed synthesis methodology in Chapter 4 to other domains where it is beneficial to generate code from English descriptions. For instance, we believe that our proposed synthesis methodology could be also useful for querying data stored in other forms (e.g., noSQL databases, XML documents, file systems) or for synthesizing simple scripts, such as if-then-else recipes or robot control commands. Another interesting direction for future work is to conduct a comprehensive survey study that compares and contrasts existing NLIDB systems developed by researchers from different areas such as natural language processing, databases and programming languages. Through out past several decades many NLIDB systems have been developed and tested on different datasets and databases. A survey that evaluates the existing NLIDB systems on a unified set of datasets and databases illustrates the strengths and weaknesses of each of these systems and shows a roadmap for future researches on this topic.

The work presented in this dissertation shows that recent advances in the field of program synthesis enable us to automatically generate programs from informal problem descriptions. Although most of the state-of-the-art program synthesis systems, including the ones presented in this dissertation, are based on either programming-by-example or programming-by-natural-language, recent researches try to develop hybrid synthesis systems which combine PBE

and PBNL approaches. We believe this hybrid methodology can improve the accuracy of synthesis systems and result in generating more complex programs.

Appendices

Appendix A

Proofs of Theorems

A.1 HADES

A.1.1 Proof of Theorem 2.1

Part 1, $\Rightarrow$. Suppose $T \equiv T'$. We show that *paths*(T) = *paths*(T'). Since $T \equiv T'$, we have height(T) = height(T') = h. Let v = root(T) and v' = root(T'). Since $T \equiv T'$, we have $L(v) = L(v') = \ell$ and $D(v) = D'(v') = d$. The proof proceeds using induction on h. For the base case, we consider $h = 1$. In this case, $V = \{v\}$, $V' = \{v'\}$, $E = E' = \emptyset$. Hence, *paths*(T) = *paths*$(T') = (\ell, d)$. For the inductive case, let $h = k + 1$ where $k \geq 1$. Let C = children(v) and let C' = children(v'). Since $T \equiv T'$, $L(C) = L'(C')$, and since T and T' are well-formed, there exists a one-to-one correspondence $f : C \rightarrow C'$ such that: $f(v_c) = v'_c$ iff$L(v_c) = L'(v'_c)$. Using this fact and condition (3) of Definition 2.4, we know that, for every $v_c \in$ children$(v) = C$, subtree$(T, v_c) \equiv$ subtree$(T', f(v_c))$. Now, observe that:

$$\text{paths}(T) = \bigcup_{v_c \in C} \{((\ell, d), p_i) \mid p_i \in \text{paths}(\text{subtree}(T, v_c))\}$$

and paths(T') is equal to:

$$\bigcup_{f(v_c) \in C'} \{((\ell, d), p'_i) \mid p'_i \in \text{paths}(\text{subtree}(T', f(v_c)))\}$$

Using the inductive hypothesis, we have $\text{paths}(T) = \text{paths}(T')$.

Part 2, $\Leftarrow$. Suppose $\mathit{paths}(T) = \mathit{paths}(T')$, and let $v = \text{root}(T)$ and $v' = \text{root}(T')$ and $L(v) = \ell, L(v') = \ell', D(v) = d, D(v') = d'$. We show that $T \equiv T'$. First, observe that $\mathit{paths}(T) = \mathit{paths}(T')$ implies $\text{height}(T) = \text{height}(T')$. The proof proceeds by induction on h. When $h = 1$, $\mathit{paths}(T) = \{(\ell, d)\}$ and $\mathit{paths}(T') = \{(\ell', d')\}$. Since $\mathit{paths}(T) = \mathit{paths}(T')$, this implies $\ell = \ell'$ and $d = d'$. Hence, $T \equiv T'$. For the inductive case, let $h = k + 1$ where $k \geq 1$. Let $C = \text{children}(v)$ and let $C' = \text{children}(v')$. Now, we have:

$$\text{paths}(T) = \bigcup_{v_i \in C} \{((\ell, d), p_i) \mid p_i \in \text{paths}(\text{subtree}(T, v_i))\}$$

and

$$\text{paths}(T') = \bigcup_{v'_i \in C'} \{((\ell', d'), p'_i) \mid p'_i \in \text{paths}(\text{subtree}(T', v'_i))\}$$

Since $\mathit{paths}(T) = \mathit{paths}(T')$, this implies $\ell = \ell'$ and $d = d'$ and, since T, T' are well-formed, for each $v_i \in C$, there must be a one-to-one correspondence $f : C \to C'$ such that $v'_i = f(v_i)$ iff $\text{paths}(\text{subtree}(T, v_i)) = \text{paths}(\text{subtree}(T', v'_i))$. Now, for any pair $(v_i, f(v_i))$, we have $L(v_i) = L'(f(v_i))$ because $\text{paths}(\text{subtree}(T, v_i)) = \text{paths}(\text{subtree}(T', v'_i))$. Note that this implies condition (2) of Definition 2.4. Furthermore, since $\text{paths}(\text{subtree}(T, v_i)) = \text{paths}(\text{subtree}(T', f(v_i)))$, the inductive hypothesis implies $\text{subtree}(T, v_i) \equiv \text{subtree}(T, v'_i)$. Hence, condition (3) of Definition 2.4 is also satisfied.

A.1.2 Proof of Path Transformer Property

Now we prove that any path transformer f returned by InferPathTrans in Section 2.4.4 must satisfy:

$$\forall p \in \mathit{inputs}(\mathcal{E}).\ (p' \in f(p) \Leftrightarrow (p, p') \in \mathcal{E})$$

Where $\mathcal{E}$ is the set of path transformation examples. Let $\Phi = \{P_1, \ldots, P_k\}$ be the set of partitions inferred by InferPathTrans at the end of Phase I (see Figure 2.2), and let each $\mathcal{P}_i$ be the triple $\langle \mathcal{E}_i, \chi_i, \phi_i \rangle$. Then the path transformer f synthesized by InferPathTrans is:

$$\lambda x.\ \{\phi_1 \rightarrow \chi_1 \oplus \ldots \oplus \phi_k \rightarrow \chi_k\}$$

We first prove that $((p, p') \in \mathcal{E}) \Rightarrow (p' \in f(p))$. First, observe that, for any k, if $\textsc{Partition}(\emptyset, \mathcal{E}, k) \neq \emptyset$, we have:

$$(\bigcup_{\mathcal{P}_i \in \Phi} \mathcal{E}_i) = \mathcal{E}$$

Hence, any $(p, p') \in \mathcal{E}$ must belong to the examples of some partition $\mathcal{P}_i$. Furthermore, our classification algorithm guarantees that, for any $p \in \mathit{inputs}(\mathcal{E}_i)$, we have $\phi_i[p/x] \equiv \mathit{true}$. Hence, we know that $\chi_i \in f(p)$. Since $\textsc{Unify}(\mathcal{E}_i) = \chi_i \neq \text{null}$, we have $\chi_i[p/x] = p'$. This implies $p' \in f(p)$.

We now prove the other direction of the property, i.e.:

$$((p \in \mathit{inputs}(\mathcal{E}) \wedge p' \in f(p)) \Rightarrow (p, p') \in \mathcal{E})$$

Suppose that $p \in \mathit{inputs}(\mathcal{E})$ and $p' \in f(p)$. Since $p' \in f(p)$, there must exist some $\mathcal{P}_i = \langle \mathcal{E}_i, \chi_i, \phi_i \rangle$ such that $\phi_i[p/x] \equiv \mathit{true}$ and $\chi_i[p/x] = p'$. Recall that classification guarantees:

$$(a) \quad \forall p \in \mathit{inputs}(\mathcal{E}_i).\ (\phi_i[p/x] \equiv \mathit{true})$$
$$(b) \quad \forall p \in (\mathit{inputs}(\mathcal{E}) - \mathit{inputs}(\mathcal{E}_i)).\ (\phi_i[p/x] \equiv \mathit{false})$$

Since $p \in \mathit{inputs}(\mathcal{E})$, this implies $p \in \mathit{inputs}(\mathcal{E}_i)$; otherwise we would have $\phi_i[p/x] \equiv \mathit{false}$. Hence, we must have $(p, p') \in \mathcal{E}_i$, which in turn implies $(p, p') \in \mathcal{E}$.

A.1.3 Proof of Theorem 2.2

Suppose $P(T) = T^*$. We will show $\mathit{paths}(T^*) = \mathit{paths}(T')$, which implies $T^* \equiv T'$ by Theorem 2.1. First, we show that, if $p' \in \mathit{paths}(T')$, then $p' \in \mathit{paths}(T^*)$. By the unambiguity criterion, there exists a unique $p \in \mathit{paths}(T)$ such that $p \sim p'$. By the correctness requirement for path transformer f, we know $p' \in f(p)$ and $p' \neq \bot$ since $p' \in \mathit{paths}(T')$. Hence, $p' \in S'$ where $S' = \{p' \mid p' \in f(p) \wedge p \in \mathit{paths}(T) \wedge p' \neq \bot\}$. Since $T^* = \textsc{Splice}(S')$ (recall code generation in Section 2.4) and Splice guarantees that $\mathit{paths}(T^*) = S'$, we also have $p' \in \mathit{paths}(T^*)$. Now, we show that, if $p^* \in \mathit{paths}(T^*)$, then $p^* \in \mathit{paths}(T')$. Since $p^* \in \mathit{paths}(T^*)$, there must exist a $p \in \mathit{paths}(T)$ such that $p^* \in f(p)$. Since $p \in \mathit{inputs}(\mathcal{E}')$, correctness of f implies there exists some $(p, p^*) \in \mathcal{E}'$. Now, suppose $p^* \notin \mathit{paths}(T')$. Since $(p, p^*) \in \mathcal{E}'$ but $p^* \notin \mathit{paths}(T')$, there are two possibilities: (i) Either $p^* = \bot$, or (ii) there is some other $(T_1, T_2) \in \mathcal{E}$ that results in p^* getting added to $\mathcal{E}'$. Now, (i) is not possible because $\mathit{paths}(T^*)$ cannot contain $\bot$, and (ii) is not possible due to the unambiguity requirement.

A.2 MITRA

A.2.1 Proof of Theorem 3.1

We first prove the soundness of $\mathcal{A}$ – i.e., if $\mathcal{A}$ accepts a column extraction program π in our DSL, then we have $\forall(\mathtt{T}, \mathcal{R}) \in \mathcal{E}.\ [\![\pi]\!]_{\{\mathtt{T.root}\},\mathtt{T}} \supseteq \mathit{column}(\mathcal{R}, i)$. Recall that $\mathcal{A}$ is the intersection of each $\mathcal{A}_j (j = 1, \cdot\cdot, |\mathcal{E}|)$ constructed using rules shown in Figure 3.11. We prove $\mathcal{A}$ is sound by proving the soundness of each $\mathcal{A}_j$. Since rule (5) in Figure 3.11 marks a state q_s to be a final state only if we have $s \supseteq \mathit{column}(\mathcal{R}, i)$ and any program π accepted by $\mathcal{A}_j$ must evaluate to the value s in a final state q_s, we have that any program π accepted by $\mathcal{A}_j$ satisfies example $\mathcal{E}_j$. Therefore, we have proved the soundness of $\mathcal{A}$ also holds.

Now, we prove the completness of $\mathcal{A}$ – i.e., if there exists a DSL program π such that we have $\forall(\mathtt{T}, \mathcal{R}) \in \mathcal{E}.\ [\![\pi]\!]_{\{\mathtt{T.root}\},\mathtt{T}} \supseteq \mathit{column}(\mathcal{R}, i)$, then π is accepted by $\mathcal{A}$. We also prove this by showing the completeness of each $\mathcal{A}_j (j = 1, \cdot\cdot, |\mathcal{E}|)$ which is constructed using rules in Figure 3.11. This can be proved because (a) the construction rules exhaustively apply all the DSL operators until no more value can be produced by any DSL program, and (b) any state q_s such that we have $s \supseteq \mathit{column}(\mathcal{R}, i)$ is marked as a final state. Therefore, we have proved the completeness of $\mathcal{A}$.

A.2.2 Proof of Theorem 3.2

Given a set of examples $\mathcal{E}$ and a table extractor ψ, assume there exists a formula φ (a boolean combination of atomic predicates) in our predicate language such that we have $\forall(\mathtt{T}, \mathcal{R}) \in \mathcal{E}.\ [\![\mathit{filter}(\psi, \lambda t.\varphi)]\!]_{\mathtt{T}} = \mathcal{R}$, now we prove

that Algorithm 3.3 returns a formula ϕ such that (a) we have $\forall(\mathtt{T}, \mathcal{R}) \in \mathcal{E}.\ [\![\mathit{filter}(\psi, \lambda t.\phi)]\!]_{\mathtt{T}} = \mathcal{R}$, and (b) formula ϕ has the smallest number of unique atomic predicates.

First, given a set Φ^* of atomic predicates and suppose there exists a formula constructed as a boolean combination of atomic predicates in Φ^* that can filter out all spurious tuples, it is obvious that Algorithm 3.3 (lines 14–16) is guaranteed to find such a formula ϕ.

Then, we show that the call to the FindMinCover procedure (given in Algorithm 3.4) at line 13 in Algorithm 3.3 returns a set Φ^* of atomic predicates such that (1) there exists a boolean combination of predicates in Φ^* that can filter out all spurious tuples, and (2) the set Φ^* is smallest. In particular, condition (1) holds because Algorithm 3.4 learns all the necessary atomic predicates for differentiating the positive examples and negative examples among all possible atomic predicates in our DSL, and conditional (2) holds due to our ILP formulation (lines 7–10).

Now, we have proved that Algorithm 3.3 returns a formula ϕ that is a valid filtering formula and has the smallest number of atomic predicates.

A.2.3 Proof of Theorem 3.3

Given a set of input-output examples $\mathcal{E}$, we show that for each example $\mathcal{E}_j$: (a) the learnt column extraction program for column i overapproximates column i in the output table, (b) the learnt table extraction program overapproximates the output table, and (c) the synthesized filter program returns

exactly the output table.

First, condition (a) holds due to THEOREM 3.1. Condition (b) also holds because our synthesis algorithm constructs the table extraction program by taking the cross-product of all column extraction programs. Finally, condition (c) holds because of THEOREM 3.2. Therefore, we conclude the proof.

A.2.4 Proof of Theorem 3.4

This theorem holds because (1) the completeness of column extraction program synthesis (as proved in THEOREM 3.1), and (2) the completness of the predicate learning algorithm (as proved in THEOREM 3.2).

A.2.5 Proof of Theorem 3.5

Our synthesis algorithm (shown in Algorithm 3.1) iterates over all possible table extraction programs (line 8). For each candidate table extraction program ψ, our algorithm learns the smallest formula ϕ (line 11) and constructs a corresponding filter program (line 13). Because our cost function θ always assigns a lower cost to a smaller formula, the learnt filter program in each iteration is guaranteed to have the lowest cost among all the filter programs that use the same table extraction program. Finally, the synthesized filter program P^* (line 15) has the lowest cost due to the update at line 14. Therefore, we conclude the proof.

Appendix B

Program Optimization in Mitra

Mitra's DSL decomposes a tree-to-table transformation task into two subproblems, namely (i) generating a set of column extractors and (ii) learning a filtering predicate. While this DSL is chosen intentionally to facilitate the synthesis task, the resulting programs may be inefficient. In particular, programs represented in this DSL first extract all possible values for each column of the table from the input tree, then generate all possible tuples as the cross-product of those columns, and finally remove all spurious tuples. The goal of the program optimization step is to apply the predicate as early as possible to avoid the generation of undesired tuples rather than filtering them out later.

The program optimization step in Mitra works as follows: Consider a synthesized program of the form $\lambda\tau.\mathit{filter}(\psi, \lambda t.\phi)$ where $\psi = \pi_1 \times \ldots \times \pi_k$. Without any optimization, this program would be implemented as:

$$
\begin{array}{l}
R \ = \ \emptyset \\
for\ n_1 \in \pi_1(T) \\
\quad for\ n_2 \in \pi_2(T) \\
\quad\quad \ldots \\
\quad\quad\quad for\ n_k \in \pi_k(T) \\
\quad\quad\quad\quad if(\phi((n_1, n_2, \ldots, n_k))) \\
\quad\quad\quad\quad\quad R \ = \ R \cup \{(n_1, n_2, \ldots, n_k)\}
\end{array}
$$

The optimizer first converts the predicate ϕ to a CNF formula $\phi = \phi_1 \wedge \ldots \wedge \phi_m$. It then generates two formulas ψ and χ such that ψ is used to guide the optimization, and χ corresponds to the remaining filtering predicate that is not handled by the optimization. Specifically, ψ, χ are both initialized to true and updated as follows:: For each clause ϕ_i of the CNF formula, we conjoin it with ψ if it is of the form $((\lambda n.\varphi_1)\ t[i]) \trianglelefteq ((\lambda n.\varphi_2)\ t[j])$ and with χ otherwise.

Now, we use formula ψ to optimize the program by finding shared paths of different column extractors. For each $\phi_k = ((\lambda n.\varphi_1)\ t[i]) \trianglelefteq ((\lambda n.\varphi_2)\ t[j] \in \psi$ and extractors π_i, π_j for columns i and j, the optimizer generates two extractors $\pi_i^{'} = \varphi_1(\pi_i)$ and $\pi_j^{'} = \varphi_2(\pi_j)$. Then, it checks whether $\pi_i^{'}$ and $\pi_j^{'}$ are semantically equivalent programs. If they are not equivalent, the optimizer removes ϕ_k from ψ and conjoins it with χ. If $\pi_i^{'}$ is equivalent to $\pi_j^{'}$ and it is a prefix of both π_i and π_j, then we can represent $\pi_i = \pi^i_{\mathit{suffix}}(\pi_i^{'})$ and $\pi_j = \pi^j_{\mathit{suffix}}(\pi_i^{'})$. This transformation allows us to share the execution of $\pi_i^{'}$ for columns i and j and has the following two benefits: First, it eliminates redundant computation, and, even more importantly, it guarantees that extracted nodes n_i and n_j satisfy predicate ψ. This kind of reasoning allows us to generate the following optimized program:

$$
\begin{array}{l}
R \ = \ \emptyset \\
for \ n_1 \in \pi_1(T) \\
\quad \ldots \\
\qquad for \ n_{ij} \in \pi_i^{'}(T) \\
\qquad\quad n_i = \pi^i_{suffix}(n_{ij}) \\
\qquad\quad n_j = \pi^j_{suffix}(n_{ij}) \\
\qquad\qquad \ldots \\
\qquad\qquad\quad for \ n_k \in \pi_k(T) \\
\qquad\qquad\qquad if(\chi((n_1, n_2, \ldots, n_k))) \\
\qquad\qquad\qquad\quad R \ = \ R \cup \{(n_1, n_2, \ldots, n_k)\}
\end{array}
$$

Observe that the optimized code has a single loop for nodes n_i and n_j rather than a nested loop.

Bibliography

[1] Automator. http://automator.us.

[2] Database schemas. goo.gl/xRMDTe.

[3] Dblp dataset. http://dblp.uni-trier.de/xml/.

[4] Imdb dataset. http://www.imdb.com/interfaces.

[5] Imdb to json. https://github.com/oxplot/imdb2json/blob/master/Readme.md.

[6] Mondial dataset. http://www.dbis.informatik.uni-goettingen.de/Mondial/.

[7] Oxygen xml editor. https://www.oxygenxml.com.

[8] Smartsheet. https://www.smartsheet.com.

[9] Treesheets. http://strlen.com/treesheets.

[10] XSL transformations 2.0. http://www.w3.org/TR/xslt20/.

[11] Yelp dataset. https://www.yelp.com/dataset_challenge.

[12] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[13] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.

[14] Bogdan Alexe, Balder TEN Cate, Phokion G Kolaitis, and Wang-Chiew Tan. Characterizing schema mappings via data examples. *TODS*, 36(4):23, 2011.

[15] Bogdan Alexe, Laura Chiticariu, Renée J Miller, and Wang-Chiew Tan. Muse: Mapping understanding and design by example. In *ICDE*, pages 10–19. IEEE, 2008.

[16] Bogdan Alexe, Balder Ten Cate, Phokion G Kolaitis, and Wang-Chiew Tan. Designing and refining schema mappings via data examples. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 133–144. ACM, 2011.

[17] Bogdan Alexe, Balder Ten Cate, Phokion G Kolaitis, and Wang-Chiew Tan. EIRENE: Interactive design and refinement of schema mappings via data examples. *VLDB*, 4(12):1414–1417, 2011.

[18] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[19] Sihem Amer-Yahia, Fang Du, and Juliana Freire. A comprehensive solution to the XML-to-relational mapping problem. In *Proceedings of*

the 6th annual ACM international workshop on Web information and data management, pages 31–38. ACM, 2004.

[20] Sihem Amer-Yahia, Fang Du, and Juliana Freire. A comprehensive solution to the xml-to-relational mapping problem. In *Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 31–38. ACM, 2004.

[21] I Androutsopoulos, G Ritchie, and P Thanisch. Masque/sql: An efficient and portable natural language query interface for relational databases. *Tech report, University of Edinburgh*, 1993.

[22] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1995.

[23] Marcelo Arenas and Leonid Libkin. XML data exchange: consistency and query answering. *Journal of the ACM (JACM)*, 55(2):7, 2008.

[24] Mustafa Atay, Artem Chebotko, Dapeng Liu, Shiyong Lu, and Farshad Fotouhi. Efficient schema-based XML-to-Relational data mapping. *Information Systems*, 32(3):458–476, 2007.

[25] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, pages 97–105, 2003.

[26] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[27] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *PLDI*, pages 218–228, 2015.

[28] David W. Barron and Christopher Strachey. Programming. *Advances in Programming and Non-Numerical Computation*, pages 49–82, 1966.

[29] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *EMNLP*, pages 1533–1544, 2013.

[30] Sonia Bergamaschi, Francesco Guerra, Matteo Interlandi, Raquel Trillo Lado, and Yannis Velegrakis. QUEST: A keyword search system for relational data based on semantic and machine learning techniques. *PVLDB*, pages 1222–1225, 2013.

[31] Philip A Bernstein and Laura M Haas. Information integration in the enterprise. *Communications of the ACM*, 51(9):72–79, 2008.

[32] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–234, 2014.

[33] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[34] Roderick Bloem, Krishnendu Chatterjee, Thomas A Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *Computer Aided Verification*, pages 140–156. Springer, 2009.

[35] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. Soda: Generating sql for business users. *PVLDB*, 5(10):932–943, 2012.

[36] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, pages 64–75. IEEE, 2002.

[37] Yuriy Brun and Michael D Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, 2004.

[38] Julien Carme, Rémi Gilleron, Aurélien Lemay, and Joachim Niehren. Interactive learning of node selecting tree transducer. *Machine Learning*, 66(1):33–67, 2007.

[39] Bob Carpenter. *Type-logical semantics*. MIT press, 1997.

[40] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In

Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 207–220. ACM, 2014.

[41] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14, 2013.

[42] Rada Chirkova, Leonid Libkin, and Juan L Reutter. Tractable XML data exchange via relations. In *Proceedings of the 20th ACM international conference on information and knowledge management*, pages 1629–1638. ACM, 2011.

[43] E. F. Codd. Seven steps to rendezvous with the casual user. In *IFIP Working Conference Data Base Management*, pages 179–200, 1974.

[44] Olivier Danvy and Michael Spivey. On barron and strachey's cartesian product function. In *ICFP*, pages 41–46, 2007.

[45] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[46] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 345–356, New York, NY, USA, 2016. ACM.

[47] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *ICSE*, 2016.

[48] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *ACM SIGMOD Record*, volume 28, pages 431–442. ACM, 1999.

[49] Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. Optimal guard synthesis for memory safety. 2014.

[50] Hong-Hai Do and Erhard Rahm. COMA: a system for flexible combination of schema matching approaches. In *VLDB*, pages 610–621, 2002.

[51] Frank Drewes and Johanna Högberg. Learning a regular tree language from a teacher. In *Developments in Language Theory*, pages 279–291. Springer, 2003.

[52] Ibrahim Dweib, Ayman Awadi, Seif Elduola Fath Elrhman, and Joan Lu. Schemaless approach of mapping XML document into Relational Database. In *Computer and Information Technology, 2008. CIT 2008. 8th IEEE International Conference on*, pages 167–172. IEEE, 2008.

[53] Jason Eisner. Learning non-isomorphic tree mappings for machine translation. In *Proceedings of Meeting on Association for Computational Linguistics*, pages 205–208, 2003.

[54] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2011.

[55] Hazem Elmeleegy, Mourad Ouzzani, and Ahmed Elmagarmid. Usage-based schema matching. In *ICDE*, pages 20–29. IEEE, 2008.

[56] Ronald Fagin, Laura M Haas, Mauricio Hernández, Renée J Miller, Lucian Popa, and Yannis Velegrakis. Clio: Schema mapping creation and data exchange. In *Conceptual modeling: foundations and applications*, pages 198–236. Springer, 2009.

[57] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[58] Ronald Fagin, Phokion G Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM Transactions on Database Systems (TODS)*, 30(1):174–210, 2005.

[59] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 422–436. ACM, 2017.

[60] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W.

Reps. Component-based synthesis for complex APIs. In *POPL*, pages 599–612, 2017.

[61] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *SIGPLAN Not.*, 50(6):229–239, June 2015.

[62] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239, 2015.

[63] Ariel Fuxman, Mauricio A Hernandez, Howard Ho, Renee J Miller, Paolo Papotti, and Lucian Popa. Nested mappings: schema mapping reloaded. In *Proceedings of the 32nd international conference on Very large data bases*, pages 67–78. VLDB Endowment, 2006.

[64] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *CAV*, pages 69–87, 2014.

[65] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. Data-guided repair of selection statements. In *ICSE*, pages 243–253. ACM, 2014.

[66] Georg Gottlob and Alan Nash. Efficient core computation in data exchange. *Journal of the ACM (JACM)*, 55(2):9, 2008.

[67] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair. In *ICSE*, pages 3–13, 2012.

[68] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *SPIN*, 2003.

[69] Barbara J. Grosz. TEAM: A transportable natural-language interface system. In *ANLP*, pages 39–45, 1983.

[70] Barbara J. Grosz, Douglas E. Appelt, Paul A. Martin, and Fernando C. N. Pereira. TEAM: an experiment in the design of transportable natural-language interfaces. *Artificial Intelligence*, 32(2):173–243, 1987.

[71] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24. ACM, 2010.

[72] Sumit Gulwani. Dimensions in program synthesis. In *FMCAD*, 2010.

[73] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[74] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.

[75] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 8–14. IEEE, 2012.

[76] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[77] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61. ACM, 2011.

[78] Sumit Gulwani and Mark Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, 2014.

[79] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. Strisynth: synthesis for live programming. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 701–704. IEEE Press, 2015.

[80] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[81] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, volume 48, pages 27–38. ACM, 2013.

[82] Carole D. Hafner. Interaction of knowledge sources in a portable natural language interface. In *COLING*, 1984.

[83] William R Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–328. ACM, 2011.

[84] William R Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328. ACM, 2011.

[85] Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. Developing a natural language interface to complex data. *TODS*, 3(2):105–147, 1978.

[86] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[87] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB'02*.

[88] Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. XParent: An efficient RDBMS-based XML database system. In *ICDE*, pages 335–336. IEEE, 2002.

[89] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *Proceedings of the*

2017 ACM International Conference on Management of Data, SIGMOD '17, pages 683–698. ACM, 2017.

[90] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[91] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[92] Manu Jose and Rupak Majumdar. Bug-assist: assisting fault localization in ansi-c programs. In *CAV*, 2011.

[93] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. *PLDI*, 2011.

[94] Jaewoo Kang and Jeffrey F Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, pages 205–216. ACM, 2003.

[95] Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *Pacific Rim International Conference on Artificial Intelligence*, pages 199–210, 2008.

[96] Rohit J. Kate and Raymond J. Mooney. Using string-kernels for learning semantic parsers. In *ACL*, 2006.

[97] Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. Learning to transform natural to formal languages. In *AAAI*, pages 1062–1068, 2005.

[98] Emanuel Kitzelmann. Analytical inductive functional programming. In *Logic-Based Program Synthesis and Transformation*, pages 87–102. Springer, 2009.

[99] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *Approaches and Applications of Inductive Programming, Third International Workshop*, pages 50–73, 2009.

[100] Emanuel Kitzelmann. A combined analytical and search-based approach for the inductive synthesis of functional programs. *KI-Künstliche Intelligenz*, 25(2):179–182, 2011.

[101] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006.

[102] Phokion G Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75. ACM, 2005.

[103] Rajasekar Krishnamurthy. *Xml-to-sql query translation*. PhD thesis, University of Wisconsin–Madison, 2004.

[104] Roland Kuhn and Renato de Mori. The application of semantic classification trees to natural language understanding. *PAMI*, 17(5):449–460, 1995.

[105] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[106] Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. *Scaling semantic parsers with on-the-fly ontology matching*, pages 1545–1556. Association for Computational Linguistics (ACL), 2013.

[107] Nada Lavrac and Saso Dzeroski. Inductive logic programming. In *WLP*, pages 146–160. Springer, 1994.

[108] Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 542–553. ACM, 2014.

[109] Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *PLDI*, page 55, 2014.

[110] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, pages 193–206, 2013.

[111] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246. ACM, 2002.

[112] Alan Leung, John Sarracino, and Sorin Lerner. Interactive and Synthesis by Example. PLDI, pages 565–574. ACM, 2015.

[113] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

[114] Yunyao Li, Huahai Yang, and H. V. Jagadish. Constructing a generic natural language interface for an XML database. In *EDBT*, pages 737–754, 2006.

[115] Percy Liang, Michael I. Jordan, and Dan Klein. Learning dependency-based compositional semantics. *Computational Linguistics*, 2013.

[116] Percy Liang and Christopher Potts. Bringing machine learning and compositional semantics together. *Annual Review of Linguistics*, 1(1):355–376, 2015.

[117] Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[118] Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[119] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *ESEC/FSE*, 2015.

[120] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *POPL*, 2016.

[121] Bill MacCartney and Christopher D Manning. An extended model of natural logic. In *IWCS*, pages 140–156, 2009.

[122] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic Schema Matching with Cupid. In *VLDB*, pages 49–58, 2001.

[123] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[124] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL System Demonstrations*, pages 55–60, 2014.

[125] Bruno Marnette, Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, Donatello Santoro, et al. ++ Spicy: an Open-Source Tool for Second-Generation Schema Mapping and Data Exchange. *VLDB*, 4:1438–1441, 2011.

[126] Edward J McCluskey. Minimization of boolean functions. *Bell Labs Technical Journal*, 35(6):1417–1444, 1956.

[127] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.

[128] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.

[129] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[130] Renée J Miller, Laura M Haas, and Mauricio A Hernández. Schema mapping as query discovery. In *VLDB*, volume 2000, pages 77–88, 2000.

[131] Scott Miller, David Stallard, Robert J. Bobrow, and Richard M. Schwartz. A fully statistical approach to natural language interfaces. In *ACL*, pages 55–61, 1996.

[132] Arnab Nandi and Philip A Bernstein. HAMSTER: using search clicklogs for schema and taxonomy matching. *VLDB*, 2(1):181–192, 2009.

[133] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *ICSE*, pages 772–781, 2013.

[134] Roland Olsson. Inductive functional programming using incremental program transformation. *Artif. Intell.*, 74(1):55–8, 1995.

[135] Peter-Michael Osera and Steve Zdancewic. Type- and example-directed program synthesis. In *PLDI*, 2015.

[136] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630, 2015.

[137] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *PLDI*, volume 47, pages 275–286. ACM, 2012.

[138] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *PLDI*, page 43, 2014.

[139] Reinhard Pichler and Vadim Savenkov. data exchange modeling tool. *Proceedings of the VLDB Endowment*, 2(2):1606–1609, 2009.

[140] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, pages 522–538, 2016.

[141] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A framework for inductive program synthesis. pages 107–126. ACM, 2015.

[142] Lucian Popa, Yannis Velegrakis, Mauricio A Hernández, Renée J Miller, and Ronald Fagin. Translating web data. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 598–609. VLDB Endowment, 2002.

[143] Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

[144] Ana-Maria Popescu, Oren Etzioni, and Henry A. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.

[145] Li Qian, Michael J Cafarella, and HV Jagadish. Sample-driven schema mapping. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 73–84. ACM, 2012.

[146] Willard V Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[147] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[148] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL*, pages 878–888, 2015.

[149] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, 2015.

[150] Mary Roth and Wang-Chiew Tan. Data Integration and Data Exchange: It's Really About Time. In *CIDR*, 2013.

[151] William C Rounds. Mappings and grammars on trees. *Theory of Computing Systems*, 4(3):257–287, 1970.

[152] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. *ACM Sigplan Notices*, 41(10):413–430, 2006.

[153] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 295–306. ACM, 2008.

[154] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of XML documents. In *International Workshop on the World Wide Web and Databases*, pages 137–150. Springer, 2000.

[155] Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Efstratios Viglas, Jeffrey Naughton, and Igor Tatarinov. A general technique for querying xml documents using a relational database system. *ACM SIGMOD Record*, 30(3):20–26, 2001.

[156] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J DeWitt, and Jeffrey F Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314. Morgan Kaufmann Publishers Inc., 1999.

[157] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings*

of the 25th International Conference on Very Large Data Bases, VLDB '99, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[158] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592, 2013.

[159] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *IJCAI*, pages 260–267, 1975.

[160] Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. Précis: From unstructured keywords as queries to structured databases as answers. *VLDB*, 2008.

[161] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827, 2016.

[162] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification*, pages 634–651. Springer, 2012.

[163] Calvin Smith and Aws Albarghouthi. MapReduce Program Synthesis. PLDI, pages 326–340. ACM, 2016.

[164] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, 2008.

[165] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[166] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[167] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[168] Michael Spivey. Functional pearls: Combinators for breadth-first search. *Journal of Functional Programming*, 10(4):397–408, 2000.

[169] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.

[170] Lappoon R Tang and Raymond J Mooney. Automated construction of database interfaces: Integrating statistical and relational learning for semantic parsing. In *EMNLP*, pages 133–141, 2000.

[171] Sandeep Tata and Guy M. Lohman. Sqak: Doing more with keywords. In *SIGMOD*, pages 889–902, 2008.

[172] Igor Tatarinov, Stratis D Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered

XML using a relational database system. In *SIGMOD*, pages 204–215. ACM, 2002.

[173] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *SIGMOD*, pages 535–548, 2009.

[174] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. *STTT*, 15(5-6):413–431, 2013.

[175] David L. Waltz. An english language query answering system for a large relational data base. *CACM*, 21(7):526–539, 1978.

[176] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM, 2017.

[177] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. In *arXiv preprint https://arxiv.org/abs/1707.01469*, To appear at OOPSLA'17, 2017.

[178] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. Fidex: Filtering spreadsheet data using examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 195–213. ACM, 2016.

[179] David H. D. Warren and Fernando C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, 1982.

[180] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[181] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *OOPSLA*, pages 19–36, 2008.

[182] Steven A. Wolfman, Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Mixed initiative interfaces for learning tasks: Smartedit talks back. In *IUI*, pages 167–174, 2001.

[183] William A Woods, Ronald M Kaplan, and Bonnie Nash-Webber. *The lunar sciences natural language information system*. Bolt, Beranek and Newman, 1972.

[184] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 508–521, New York, NY, USA, 2016. ACM.

[185] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *CoRR*, abs/1711.04001, 2017.

[186] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26, 2017.

[187] Ling Ling Yan, Renée J Miller, Laura M Haas, and Ronald Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD*, volume 30, pages 485–496. ACM, 2001.

[188] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.

[189] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. Xrel: a path-based approach to storage and retrieval of xml documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.

[190] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 20. ACM, 2014.

[191] John M. Zelle and Raymond J. Mooney. Learning semantic grammars with constructive inductive logic programming. In *AAAI*, pages 817–822, 1993.

[192] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI*, pages 1050–1055, 1996.

[193] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI*, 1996.

[194] Sai Zhang and Yuyin Sun. Automatically synthesizing sql queries from input-output examples. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 224–234. IEEE, 2013.

[195] Sai Zhang and Yuyin Sun. Automatically synthesizing sql queries from input-output examples. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 224–234. IEEE, 2013.

[196] Moshé M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *VLDB*, 1975.